Облачные архитектуры

РАЗРАБОТКА УСТОЙЧИВЫХ И ЭКОНОМИЧНЫХ ОБЛАЧНЫХ ПРИЛОЖЕНИЙ

Том Лащевски, Камаль Арора, Эрик Фарр, Пийюм Зонуз

ПИТЕР®

Packt›

Cloud Native Architectures

Design high-availability and cost-effective applications for the cloud

Tom Laszewski
Kamal Arora
Erik Farr
Piyum Zonooz

Packt›

BIRMINGHAM - MUMBAI

Облачные архитектуры

РАЗРАБОТКА УСТОЙЧИВЫХ И ЭКОНОМИЧНЫХ ОБЛАЧНЫХ ПРИЛОЖЕНИЙ

Том Лащевски
Камаль Арора
Эрик Фарр
Пийюм Зонуз

ПИТЕР®
Санкт-Петербург · Москва · Минск
2022

Том Лащевски, Камаль Арора, Эрик Фарр, Пийюм Зонуз

Облачные архитектуры: разработка устойчивых и экономичных облачных приложений

Серия «Библиотека программиста»

Перевел с английского *А. Павлов*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*А. Саидов*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Н. Рощина*
Верстка	*Г. Блинов*

ББК 32.988.02
УДК 004.738.5

Том Лащевски, Камаль Арора, Эрик Фарр, Пийюм Зонуз

О-16 Облачные архитектуры: разработка устойчивых и экономичных облачных приложений. — СПб.: Питер, 2022. — 320 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1588-4

Облачные вычисления — это, пожалуй, наиболее революционная разработка в IT со времен виртуализации. Облачно-ориентированные архитектуры обеспечивают большую гибкость по сравнению с системами предыдущего поколения. В этой книге продемонстрированы три важнейших аспекта развертывания современных cloud native архитектур: организационное преобразование, модернизация развертывания, паттерны облачного проектирования.

Книга начинается с краткого знакомства с облачно-ориентированными архитектурами — на примерах объясняется, какие черты им присущи, а какие нет. Вы узнаете, как организуется внедрение и разработка облачных архитектур с применением микросервисов и бессерверных вычислений как основ проектирования. Далее вы изучите такие столпы облачно-ориентированного проектирования, как масштабируемость, оптимизация издержек, безопасность и способы достижения безупречной эксплуатационной надежности. В заключительных главах будет рассказано о различных общедоступных архитектурах cloud native — от AWS и Azure до Google Cloud Platform.

Прочитав эту книгу, вы освоите приемы, необходимые для перехода на облачно-ориентированные архитектуры с учетом требований вашего бизнеса. Вы также узнаете о перспективных тенденциях в сфере облачных сервисов и векторах развития облачных провайдеров.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1787280540 англ.

ISBN 978-5-4461-1588-4

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 2. Тел.: +78127037373.
Дата изготовления: 08.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 30.07.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 500. Заказ 0000.
Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт».
109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. I, ком. 6.3-23Н.

Краткое содержание

Оглавление

Предисловие

Цель книги — определить область практического применения облачных вычислений. Это исследование их возможностей, отвечающее на множество вопросов *«почему»* и *«как»*.

Основная цель облачных технологий состоит в повышении мощности облачных вычислений и возможности заставить приложения работать в облаке так, как они могут это делать на локальных машинах. Книга будет полезна всем, кто имеет отношение к облачным вычислениям, включая разработчиков, конечных пользователей и, таким образом, весь бизнес в целом. Целью ИТ-разработчиков всегда должно быть удовлетворение потребностей бизнеса. Издание поможет ИТ-специалистам составить эффективные планы для достижения этой цели.

Рассмотрим преимущества перехода к облачным технологиям.

- *Производительность*. Предоставляемый по умолчанию доступ к встроенным функциям облачных сервисов обеспечивает более высокую производительность по сравнению с той, что возможна при использовании внешних функций. Например, вы можете пользоваться системой ввода-вывода, которая работает с функциями автоматического масштабирования и балансировки нагрузки.
- *Эффективность*. Благодаря своим особенностям облачно-ориентированные приложения позволяют более эффективно использовать базовые ресурсы. Это приводит к повышению производительности и/или снижению эксплуатационных расходов.
- *Стоимость*. Более эффективные приложения обычно дешевле в эксплуатации. Облачные провайдеры ежемесячно присылают счета в зависимости от количества потребляемых ресурсов, поэтому, если выполнить больше операций с меньшими затратами, можно существенно сэкономить.
- *Масштабируемость*. Поскольку вы используете приложения, оптимизированные для облачных интерфейсов, то получаете прямой доступ к функциям автоматического масштабирования и балансировки нагрузки облачной платформы.

Чтобы воспользоваться преимуществами облачной платформы, в том числе платформ IaaS (таких как AWS), вы должны проектировать приложения так, чтобы они не были привязаны к конкретному физическому ресурсу. Например, если вы обращаетесь к вводу-выводу напрямую с такой платформы, как Linux, вам необходимо получить доступ к уровню абстракции облака или к его собственным API.

Облака *могут* обеспечить уровень абстракции или виртуализации между приложением и базовыми физическими (или виртуальными) ресурсами независимо от того, разработаны они для облака или нет. Но этого недостаточно. Если вы собираетесь

пользоваться облачными технологиями, вам придется напрямую управлять нужными ресурсами.

Когда архитектура учитывается при проектировании, разработке и развертывании приложения, использование базовых облачных ресурсов может быть на 70 % эффективнее. В результате увеличение эффективности облачных вычислений позволяет сэкономить деньги. Вы платите за используемые ресурсы, поэтому приложения, которые расходуют их наиболее эффективно, работают быстрее и обходятся дешевле в финансовом плане.

Cloud native — это не только приведение кода в соответствие с особенностями конкретного облака, но и изменение подхода к архитектурному проектированию. Такие ориентированные на облако архитектуры могут быть автомасштабированными, распределенными, без сохранения состояния и слабо связанными — и это лишь некоторые из характеристик. Если вы хотите сделать приложения облачно-ориентированными, то прежде, чем начинать рефакторинг кода, постарайтесь переосмыслить его архитектуру.

Затратен ли новый подход к архитектуре приложений по ресурсам и деньгам, рискованнее ли он? Да. Тем не менее соотношение «риск/вознаграждение», как правило, склоняется в сторону вознаграждения, если срок службы приложения составляет от 10 до 15 лет (что справедливо для большинства предприятий). *Усилия по реорганизации и рефакторингу приложения с долгосрочным использованием окупятся многократно.*

При переносе приложений в облако аргументы более убедительны в отношении подхода cloud native. Для большинства приложений, выбранных для перемещения в облако, преимущества перевешивают затраты, но учитывая, что затраты на рефакторинг в 30 раз больше, чем на простое повторное размещение, предприятия неохотно решаются полностью перейти к такому подходу.

Таким образом, возникает еще один процесс обучения — он будет подобен тому, в ходе которого мы разрабатывали приложения на основе API, ориентированных на такие платформы, как Unix/Linux. Нам пришлось исправлять ошибки, прежде чем мы добились успеха. Подозреваю, что и теперь мы будем следовать той же схеме. Через несколько лет облачная среда станет лучшим из того, что сможет предложить ИТ-индустрия. Однако этого не произойдет, пока мы не ошибемся еще несколько раз. Некоторые вещи остаются неизменными.

Итак, облачно-ориентированные приложения, которые используют облачные архитектуры, — это наш путь? Судя по тому, что вы читаете предисловие к данной книге, вы уже уверены, что это так. Прочитав всю книгу, вы только убедитесь в этом. Когда вы развернете свою первую или вторую архитектуру cloud native, вы на практике увидите, что это так.

Дэвид Линтикум (David Linthicum), главный специалист по облачной стратегии, Deloitte Consulting LLP

В тот момент, когда я пишу эти строки (летом 2018 года), к использованию облачных вычислений уже перешли многие организации. Размещая приложения в облаке, компании могут сократить расходы и перестать зависеть от ресурсов, не связанных с добавленной стоимостью, таких как серверы и сети. Неудивительно, что, по данным RightScale, почти 90 % компаний каким-то образом используют облачные ресурсы.

Однако применение облака просто как дешевого внешнего или виртуализированного центра обработки данных все еще оставляет нерешенной значительную часть облачных вычислений. Применив подход cloud native к системам и архитектурам приложений, компании могут воспользоваться преимуществами гибких и практически неограниченных вычислительных мощностей, систем хранения и сетевых возможностей современного общедоступного облака и тем самым стать намного более полезными для своих клиентов.

Что представляют собой приложения, оптимизированные для выполнения в облаке (cloud native)? В первую очередь они определяются возможностью автоматически масштабироваться при увеличении и снижении нагрузки. Сегодня большинство компаний сосредотачивают максимальное количество серверов в своих центрах обработки данных в ожидании большой нагрузки, а затем следят за тем, чтобы в среднем загрузка их ЦП оставалась на одном уровне. Благодаря использованию функций автоматического масштабирования в облаке приложения могут разрастаться и сокращаться по мере необходимости без вмешательства человека. Если вам нужна повышенная пропускная способность — пожалуйста; если в ней пропала необходимость, то вы ею не пользуетесь и не платите за нее.

Облачное приложение устойчиво к ошибкам и сбоям. Если по какой-то причине аппаратное обеспечение — сервер или маршрутизатор — выходит из строя или в базе данных происходит катастрофическая ошибка, облачное приложение должно обнаружить ошибку и исправить ее, возможно, путем создания нового инстанса[1] на другой стойке или даже в другом облачном дата-центре в другом регионе.

Вычисления cloud native ускоряют процессы в ИТ-сфере, что дает о себе знать и в бизнесе, который обслуживается ИТ, а это позволяет специалистам быстрее и с меньшими усилиями внедрять подходы и решения на основе Agile и DevOps. Это дает ИТ-специалистам возможность чаще внедрять новые проекты — до нескольких раз в день — и использовать для большей надежности конвейеры автоматического тестирования. Появляется возможность экспериментировать в облаке, пользуясь машинным обучением и расширенной аналитикой, без необходимости принимать решения о дорогостоящих закупках. Благодаря ускорению в ИТ-сфере компании могут предоставлять своим клиентам более качественные и быстрые решения, привлекать новых клиентов за счет глобальной доступности облака и, возможно, даже менять свои основные бизнес-модели.

---

[1] Сервера в терминологии AWS. — *Примеч. пер.*

Существует множество новых технологий — контейнеров, API-шлюзов, менеджеров событий, инструментов бессерверной обработки данных, — которые при правильной реализации могут дать вам преимущества облачных вычислений. Однако переход к cloud native подразумевает не только технологические, но и организационные и культурные изменения. В частности, переход от каскадной модели развития к гибким (Agile) и бережливым (Lean) подходам, внедрение непрерывной интеграции/непрерывной поставки позволят понять, что такое вообще облачная архитектура.

Внедрение облачных методологий и архитектур имеет множество преимуществ, включая повышение потенциала информационных технологий и, что более важно, эспоненциальное расширение инновационных возможностей для вашего бизнеса. Это делает облачно-ориентированные вычисления неотъемлемым и ведущим фактором внедрения современных облаков.

Настоятельно рекомендую вам прочитать эту книгу. В ней вы найдете примеры создания приложений, которые используют облако по назначению, и откроете новый мир продуманных инновационных решений.

Миха Краль (Miha Kralj), генеральный директор,
Cloud Native Architecture, Accenture LLP

Об авторах

Том Лащевски (Tom Laszewski) — ведущий специалист по облачным технологиям, помогавший независимым разработчикам ПО, системным администраторам, создателям стартапов и корпоративным клиентам модернизировать ИТ-системы и разрабатывать инновационные программные решения. Сегодня он возглавляет группу специалистов, отвечающих за стратегию трансформации бизнеса и ИТ ключевых клиентов AWS. Многие из этих клиентов стремятся к модернизации облачных вычислений и цифровой трансформации с использованием архитектур cloud native.

Камаль Арора (Kamal Arora) — автор книг и ведущий специалист с более чем 15-летним опытом работы в сфере ИТ. В настоящее время трудится в Amazon Web Services и возглавляет многопрофильную команду высококвалифицированных архитекторов решений, которые помогают партнерам и корпоративным клиентам переходить в облако. Камаль активно интересуется последними инновациями в облаке и пространстве AI/ML, а также их влиянием на наше общество и повседневную жизнь.

Спасибо моей жене Пунам Ароре за безусловную поддержку и моим близким — Киран, Радживу, Нише, Аарини и Риан — за их постоянную заботу и поддержку. И последнее, но не менее важное: мы скучаем по тебе, папа!

Эрик Фарр (Erik Farr) — ведущий специалист с более чем 18-летним опытом работы в ИТ-индустрии. Он участвовал в разработке передовых облачных технологий и различных корпоративных архитектур, работал с крупнейшими компаниями и системными интеграторами. Сегодня в Amazon Web Services он возглавляет команду опытных архитекторов, которые помогают глобальным партнерам и системным интеграторам создавать собственные облачные архитектуры масштаба предприятия. До работы в AWS Эрик сотрудничал с Capgemini и The Walt Disney Company, которые всегда стремились к созданию чего-то нового для клиентов.

Хотел бы сказать спасибо моей замечательной семье — Стейси, Фэйт и Сидни — за поддержку.

Пийюм Зонуз (Piyum Zonooz) — архитектор решений для глобальных партнеров в Amazon Web Services, где он работает с компаниями из различных отраслей, помогая внедрять облачные технологии и реорганизовывать продукты, делая их полностью облачными. Он возглавлял проекты по анализу TCO, проектированию инфраструктуры, внедрению DevOps и полной трансформации бизнеса. До работы в AWS Пийюм был ведущим архитектором в рамках Accenture Cloud Practice, где руководил крупномасштабными проектами по внедрению облачных технологий. Пийюм окончил Иллинойсский университет в Урбане-Шампейне.

О научном редакторе

Санджив Джайсвал (Sanjeev Jaiswal) — выпускник CUSAT, специалист в области компьютерных технологий, имеет десятилетний опыт практической работы. В основном использует Perl, Python, AWS и GNU/Linux. В сфере его интересов — пентестирование, проверка исходного кода, проектирование и реализация механизмов защиты в AWS и проектах облачной безопасности. Он также изучает DevSecOps и автоматизацию безопасности. В свободное время Санджив читает лекции студентам инженерных специальностей и ИТ-специалистам.

Он написал книгу *Instant PageSpeed Optimization* и выступил соавтором книги *Learning Django Web Development*.

Хочу поблагодарить свою жену Шалини Джайсвал за то, что всегда была рядом, и своих друзей Ранджана, Ритеша, Микки, Шанкара и Сантоша за их постоянную поддержку.

Введение

Эта книга поможет вам понять основные этапы проектирования, необходимые для построения масштабируемых систем. Вы узнаете, как эффективно планировать ресурсы и технологические стеки для достижения большей безопасности и отказоустойчивости. При этом вы изучите основные архитектурные принципы на реальных примерах. В этой книге применен практический подход с примерами из реальной жизни, которые помогут вам при разработке облачных приложений и эффективном переносе вашего бизнеса в облако.

Для кого эта книга

Эта книга предназначена для архитекторов программного обеспечения, которые заинтересованы в разработке отказоустойчивых, масштабируемых и высокодоступных облачных приложений.

Структура издания

Глава 1 «Введение в архитектуру cloud native». В этой главе рассматриваются достоинства и недостатки архитектур cloud native, связанные с ними мифы, а также возможные сложности работы в облаке.

В *главе 2 «Процесс перехода в облако»* говорится о том, что значит работать в облаке. Данное явление будет рассмотрено с разных сторон, в том числе будет рассказано о том, как выполнить миграцию из локальной или существующей среды в облако.

Глава 3 «Разработка приложений cloud native» подробно рассматривает разработку архитектур cloud native с использованием микросервисов и бессерверных вычислений.

В *главе 4 «Как выбрать технологический стек»* рассматривается общая технология, которая используется для создания архитектур cloud native, от приложений с открытым исходным кодом до лицензионного программного обеспечения. Здесь мы будем исследовать торговые площадки, которые могут быть использованы для потребления ресурсов в облаке. Наконец, обсудим процесс закупок и модели лицензирования, распространенные в облаке.

В *главе 5 «Масштабируемость и доступность»* рассказывается о доступных инструментах/функциях и стратегиях, которые можно использовать при разработке систем cloud native для их масштабирования и обеспечения высокой доступности. В этой главе будет рассмотрено, как работают эти инструменты и/или функции, как они используют приложения cloud native и как их развертывать.

В *главе 6 «Безопасность и надежность»* обсуждаются модели безопасности, функции, доступные в облаке, подводные камни и лучшие практики, связанные с безопасностью ИТ-систем. В ней также будет рассмотрено, как работают функции безопасности и как они могут помочь пользователям облака развернуть более защищенную среду.

Глава 7 «Оптимизация затрат» описывает модель ценообразования для облачных сред. Мы обсудим подходы к оценке затрат, а также различия между старой и облачной моделями.

В *главе 8 «Эксплуатация облачных сервисов»* рассказывается об инструментах, процедурах и моделях, обеспечивающих непрерывную работу сред, развернутых в облаке. В этой главе будут изложены организационные модели, стратегии управления и шаблоны развертывания для поддержки работоспособности систем.

Глава 9 «Amazon Web Services» фокусируется на том, чтобы дать представление о возможностях разработки облачных приложений Amazon Web Services.

Глава 10 «Microsoft Azure» посвящена рассмотрению возможностей разработки облачных приложений Microsoft Azure.

Глава 11 «Google Cloud Platform» посвящена ознакомлению с возможностями разработки облачных приложений на Google Cloud Platform.

В *главе 12 «А что же дальше? Тенденции развития архитектуры облачных приложений»* приведен анализ будущих тенденций и ожиданий от различных облачных провайдеров, работающих на рынке.

Как получить максимальную пользу от книги

1. При чтении книги будет полезен опыт работы с архитектурой программного обеспечения.
2. Следует внимательно изучить все примеры и инструкции, которые приведены в соответствующих местах.

Загрузка файлов примеров кода

Вы можете загрузить файлы примеров кода для этой книги на сайте GitHub по адресу github.com/PacktPublishing/Cloud-Native-Architectures.

Условные обозначения

В этой книге используется ряд условных обозначений.

`КодВТексте` обозначает фрагменты программного кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, ввод пользователя и обработчики Twitter. Например: «Для обработчика укажите значение `lambda_function.lambda_handler`».

Блок кода обозначается следующим образом:

```
print('Loading function') def respond(err, res=None): return {
'statusCode': '400' if err else '200', 'body': err if err else res,
'headers': { 'Content-Type': 'application/json', }, }
```

Когда мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются полужирным шрифтом:

```
print('Loading function')
def respond(err, res=None):
    return {
        'statusCode': '400' if err else '200',
        'body': err if err else res,
        'headers': {
          'Content-Type': 'application/json',
        },
    }
```

Курсивом выделены новые термины и важные понятия.

Таким шрифтом выделены элементы интерфейса, которые вы видите на экране, например, названия пунктов меню или диалоговых окон. Вот пример: «Щелкните на названии API Бессерверная служба погоды, чтобы войти в его конфигурацию».

Так выделены предупреждения или важные заметки.

Советы и хитрости обозначены таким образом.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Введение в архитектуру cloud native

Распространение облачных вычислений привело к появлению новой парадигмы в разработке, внедрении и обслуживании компьютерных систем. Хотя у этой парадигмы существует много разных названий, наиболее часто используемое — *«архитектуры cloud native»*. В книге мы рассмотрим, что такое архитектуры cloud native, почему они являются новым подходом и бывают разными и как внедряются во многих глобальных компаниях. Как следует из названия, речь идет об облаке и использовании сервисов облачных провайдеров для проектирования этих архитектур с целью решения бизнес-задач новыми, надежными и безопасными способами. Цель этой главы состоит в том, чтобы объяснить и определить, что представляют собой архитектуры cloud native, а также дать представление об их плюсах, минусах и связанных с ними мифах. Мы рассмотрим, что значит cloud native, разберемся с характеристиками и компонентами, которые требуются для этого типа архитектуры, и оценим шаги, которые должна предпринять компания для получения развитой (зрелой) модели.

Что такое архитектуры cloud native

Если вы спросите 100 человек, каково определение термина cloud native, то получите 100 разных ответов. Почему так много? Во-первых, сами облачные вычисления не стоят на месте, поэтому определения, предложенные несколько лет назад, возможно, не совсем соответствуют текущему состоянию облака. Во-вторых, архитектуры cloud native — это совершенно новая парадигма, использующая новые методы решения бизнес-задач, которые обычно могут быть достигнуты только в масштабе облачных вычислений. Наконец, определения могут сильно различаться в зависимости от того, какого специалиста вы просите их дать: архитектора, разработчика, администратора или руководителя, принимающего решения. Итак, что же такое cloud native?

Начнем с общепринятого определения облачных вычислений в соответствии с AWS: *«Облачные вычисления — это предоставление вычислительной мощности, хранилища базы данных, приложений и других ИТ-ресурсов по требованию через платформу облачных сервисов по Интернету с оплатой по факту»*.

Таким образом, в своей основной форме *cloud native* означает использование сервисов облачных вычислений для разработки решения, однако это только часть того,

что требуется для перехода на облачные технологии. *Cloud native* — это гораздо больше, чем просто использование базовой облачной инфраструктуры, даже если это наиболее востребованная из доступных услуг.

Автоматизация и разработка приложений также играют важную роль в этом процессе. Облако, разработанное на основе API, позволяет реализовывать чрезвычайно широкую автоматизацию и масштабирование, чтобы не только создавать инстансы или конкретные системы, но и развертывать весь корпоративный ландшафт без участия человека. Наконец, критически важным компонентом при создании облачной архитектуры является подход, используемый для разработки конкретного приложения. Системы, созданные с помощью лучших облачных сервисов и развернутые с широкой автоматизацией, могут не приносить желаемых результатов, если логика приложения не учитывает новый масштаб, в котором ему, возможно, придется работать.

Определение модели зрелости cloud native

Нет единственно правильного ответа на вопрос о том, что представляет собой архитектура cloud native, так как к ней могут быть отнесены многие типы архитектур. Используя три принципа, или оси, проектирования — сервисы cloud native, проектирование, ориентированное на приложения, и автоматизацию, — можно оценить большинство систем по уровню зрелости cloud native. Кроме того, эти принципы постоянно расширяются по мере разработки новых технологий, методов или шаблонов проектирования, и поэтому архитектуры cloud native будут продолжать развиваться. Мы, авторы этой книги, считаем, что облачные архитектуры формируются эволюционным путем и соответствуют модели зрелости возможностей (https://ru.wikipedia.org/wiki/Capability_Maturity_Model). В книге архитектуры cloud native будут описываться с помощью модели *Cloud Native Maturity Model (CNMM)* в соответствии с изложенными принципами проектирования (рис. 1.1).

Ось 1. Облачные сервисы

Чтобы разобраться, как система связана с CNMM, важно выявить компоненты архитектуры cloud native. По определению, cloud native требует использования облачных сервисов. У каждого провайдера облачных вычислений будет свой набор сервисов, причем самые продвинутые обеспечивают самый богатый набор функций. Объединение этих сервисов, от базовых строительных блоков до самых передовых технологий, определит, насколько сложна архитектура cloud native на оси облачных сервисов (рис. 1.2).

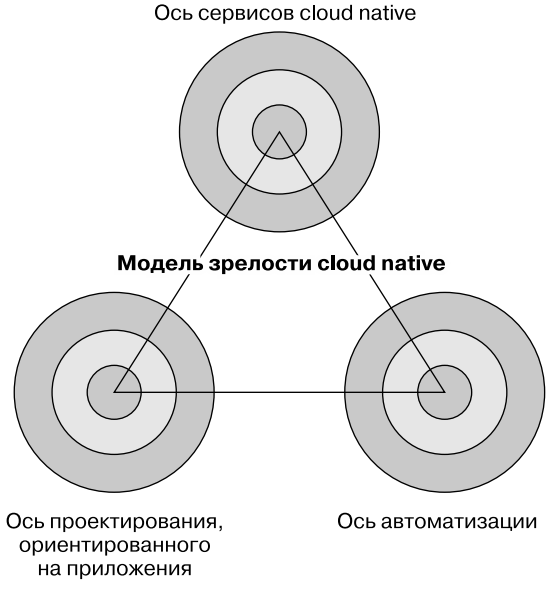


Рис. 1.1

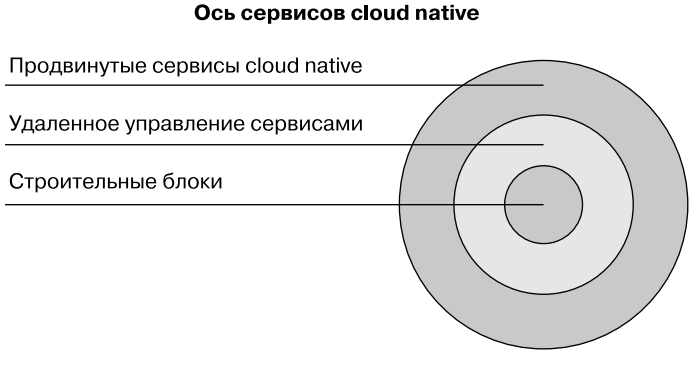


Рис. 1.2

Сервисы поставщика развитого облака

Amazon Web Services (AWS) часто называют самой продвинутой облачной платформой (на момент написания книги). На рис. 1.3 показаны все сервисы, которые AWS может предложить, от базовых строительных блоков до предложений по удаленному управлению и использованию продвинутых сервисов платформы.

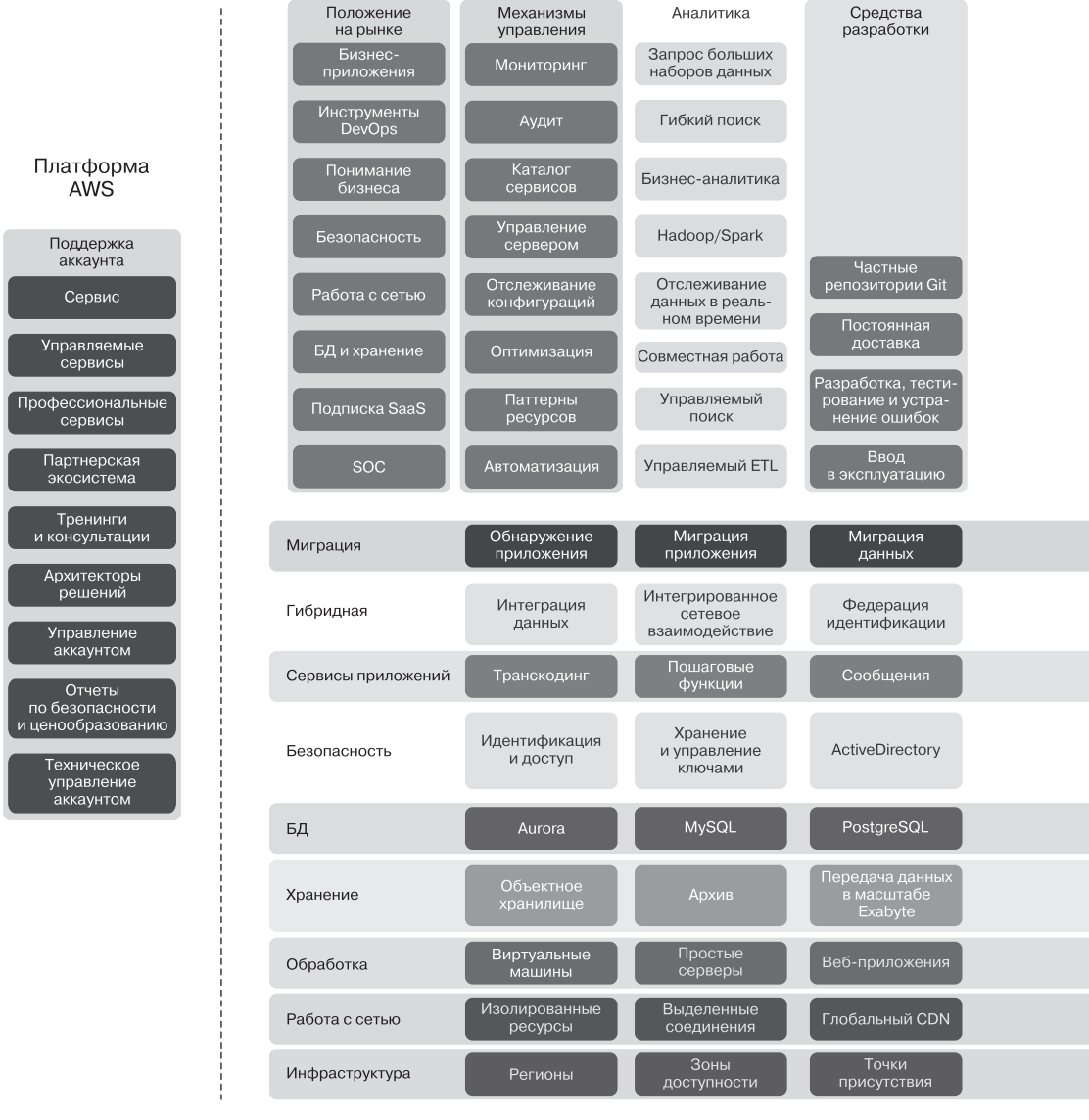
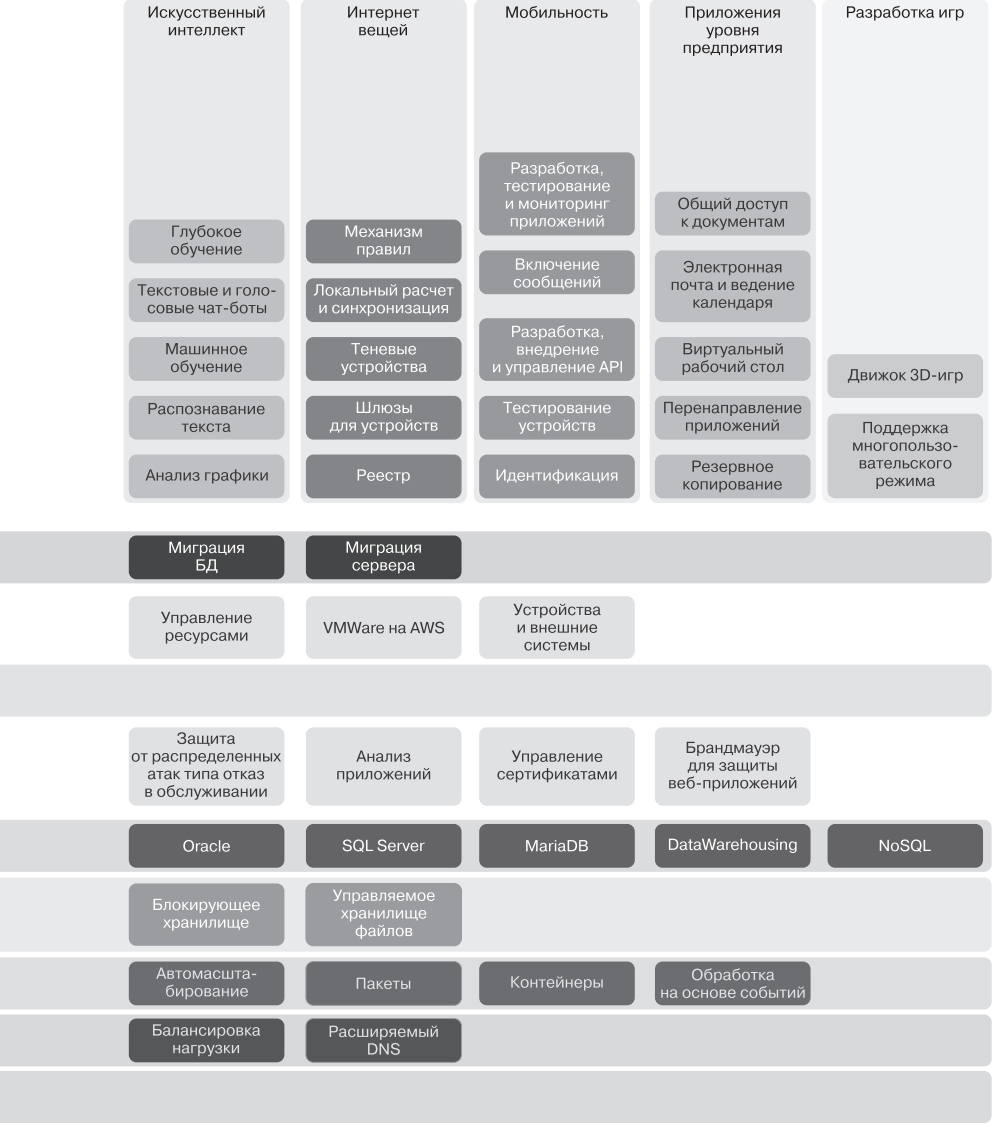

Платформа AWS
Поддержка аккаунта
Сервис
Управляемые сервисы
Профессиональные сервисы
Партнерская экосистема
Тренинги и консультации
Архитекторы решений
Управление аккаунтом
Отчеты по безопасности и ценообразованию
Техническое управление аккаунтом
Положение на рынке
Бизнес-приложения
Инструменты DevOps
Понимание бизнеса
Безопасность
Работа с сетью
БД и хранение
Подписка SaaS
SOC
Механизмы управления
Мониторинг
Аудит
Каталог сервисов
Управление сервером
Отслеживание конфигураций
Оптимизация
Паттерны ресурсов
Автоматизация
Аналитика
Запрос больших наборов данных
Гибкий поиск
Бизнес-аналитика
Hadoop/Spark
Отслеживание данных в реальном времени
Совместная работа
Управляемый поиск
Управляемый ETL
Средства разработки
Частные репозитории Git
Постоянная доставка
Разработка, тестирование и устранение ошибок
Ввод в эксплуатацию
Миграция
Обнаружение приложения
Миграция приложения
Миграция данных
Гибридная
Интеграция данных
Интегрированное сетевое взаимодействие
Федерация идентификации
Сервисы приложений
Транскодинг
Пошаговые функции
Сообщения
Безопасность
Идентификация и доступ
Хранение и управление ключами
ActiveDirectory
БД
Aurora
MySQL
PostgreSQL
Хранение
Объектное хранилище
Архив
Передача данных в масштабе Exabyte
Обработка
Виртуальные машины
Простые серверы
Веб-приложения
Работа с сетью
Изолированные ресурсы
Выделенные соединения
Глобальный CDN
Инфраструктура
Регионы
Зоны доступности
Точки присутствия

Рис. 1.3


Искусственный интеллект
Интернет вещей
Мобильность
Приложения уровня предприятия
Разработка игр
Разработка, тестирование и мониторинг приложений
Общий доступ к документам
Глубокое обучение
Механизм правил
Включение сообщений
Электронная почта и ведение календаря
Текстовые и голосовые чат-боты
Локальный расчет и синхронизация
Разработка, внедрение и управление API
Виртуальный рабочий стол
Движок 3D-игр
Машинное обучение
Теневые устройства
Тестирование устройств
Перенаправление приложений
Распознавание текста
Шлюзы для устройств
Идентификация
Резервное копирование
Поддержка многопользовательского режима
Анализ графики
Реестр
Миграция БД
Миграция сервера
Управление ресурсами
VMWare на AWS
Устройства и внешние системы
Защита от распределенных атак типа отказ в обслуживании
Анализ приложений
Управление сертификатами
Брандмауэр для защиты веб-приложений
Oracle
SQL Server
MariaDB
DataWarehousing
NoSQL
Блокирующее хранилище
Управляемое хранилище файлов
Автомасштабирование
Пакеты
Контейнеры
Обработка на основе событий
Балансировка нагрузки
Расширяемый DNS

Компоненты облачных сервисов

Независимо от уровня зрелости, поставщик облачных вычислений всегда предлагает такие возможности, как вычисления, хранение, сетевое взаимодействие и мониторинг. В зависимости от уровня зрелости облака организации и команды, разрабатывающей систему, стоит начать переход в cloud native, используя эти строительные блоки базовой инфраструктуры. Инстансы виртуального сервера, блочное дисковое хранилище, хранилище объектов, волоконно-оптические линии и VPN, балансировщики нагрузки, мониторинг облачного API и мониторинг инстансов — это все типы строительных блоков, которые клиент будет использовать в начале работы с облаком. Аналогично компонентам, доступным в существующем локальном центре обработки данных, эти сервисы позволят командам разработчиков начать знакомство с приложениями в облаке. Применение данных сервисов — это минимум, необходимый для разработки архитектуры cloud native и отражающий относительно низкий уровень на оси сервисов cloud native.

Зачастую компания выбирает миграцию существующего приложения в облако и выполняет ее по модели «поднять и переместить». В этом случае она просто перенесет стек приложений и окружающие компоненты в облако, никак не изменяя дизайн, технологии или архитектуру компонентов. В таких миграциях используются только основные строительные блоки, предлагаемые облаком, поскольку они существуют и в локально затребованных точках клиента. Хотя это низкий уровень зрелости, он важен, так как позволяет получить опыт работы с облаком. Даже при использовании строительных блоков облачных сервисов команда разработчиков быстро добавит свои собственные ограничительные правила, политики и соглашения об именах, чтобы изучить более эффективные методы решения проблем безопасности, развертывания, работы в сети и реализации прочих основных требований к облачно-ориентированным системам.

Одними из основных результатов, которые компания получит на этом этапе зрелости, являются понимание основных свойств облака и того, как эти свойства влияют на методы проектирования, — например, как соотносятся горизонтальное и вертикальное масштабирование, как эти архитектурные решения сказываются на финансовых расходах и как их эффективно реализовать. Кроме того, это позволяет понять, как работает выбранный поставщик облачных сервисов, как он группирует свои сервисы в определенных местах, а также как эти группы взаимодействуют для обеспечения высокой доступности и аварийного восстановления с помощью существующей архитектуры. Наконец, изучение облачного подхода к хранилищу и возможность передачи обработки в облачные сервисы, которые эффективно и естественно масштабируются на платформе, — критически важный подход к проектированию архитектур. Несмотря на то что использование строительных блоков облачных сервисов говорит об относительно низком уровне зрелости, оно очень важно для компаний, которые только начинают свою деятельность в облачной инфраструктуре.

Предложения поставщиков, предоставляющих облачные сервисы

Недифференцированная рутинная работа — так часто можно описать ситуацию, когда время, усилия, ресурсы или деньги затрачиваются на выполнение задач, которые не приносят прибыли компании. Термин *«недифференцированный»* означает, что не существует методов различить разные способы выполнения данной работы. *Рутинной работой* можно назвать решение сложных технологических и эксплуатационных задач, которые, если все сделано правильно, никто никогда не распознает, а их неправильное выполнение может привести к катастрофическим последствиям для бизнес-операций. Эта фраза в совокупности означает, что компания решает сложные задачи, неправильное выполнение которых может повлиять на бизнес, но в то же время ее специалисты не понимают, как все сделать правильно, что не только не приносит пользы бизнесу, но и может легко ему навредить.

К сожалению, такое положение вещей характерно для подавляющего большинства задач ИТ-отделов в корпоративных компаниях и в то же время является важным аргументом за использование облака. Поставщики облачных услуг разбираются в технологических инновациях и эксплуатации крупномасштабных систем так хорошо, как в большинстве компаний и не мечтали. Таким образом, имеет смысл только то, что поставщики облачных услуг усовершенствовали свои сервисы, включив в них удаленно управляемые решения, которые находятся под их полным контролем, а потребителю нужно лишь разработать бизнес-логику или данные, развертываемые в сервисе. Это дает возможность переложить выполнение недифференцированной рутинной работы на поставщика облачных услуг и позволяет компании выделять значительно больше ресурсов для увеличения ценности бизнеса (его отличия от конкурирующих продуктов).

Как мы уже видели, существует множество комбинаций облачных сервисов, которые можно использовать для проектирования собственных архитектур облаков с помощью только базовых строительных блоков и шаблонов. Но, когда команда разработчиков станет лучше понимать, какие сервисы имеются у выбранного поставщика облачных вычислений, и более обдуманно подходить к их выбору, она, несомненно, захочет использовать более усовершенствованные облачные сервисы. Продвинутые поставщики облачных решений будут предлагать сервисы, которые способны заменить компоненты, требующие недифференцированной рутинной работы. В число удаленно управляемых решений, которые предлагают поставщики облачных услуг, входят следующие:

- базы данных;
- Hadoop[1];

[1] Свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределенных программ, работающих на кластерах из сотен и тысяч узлов. — *Примеч. ред.*

- службы каталогов;
- балансировщики нагрузки;
- системы кеширования;
- хранилища данных;
- репозитории кода;
- инструменты автоматизации;
- поисковая система Elastic.

Еще одна важная характеристика этих сервисов — гибкость, которую они привносят в решение. Если система спроектирована для использования данных инструментов, но управляется командой эксплуатации, то зачастую процесс предоставления виртуального инстанса, конфигурации, настройки и обеспечения безопасности проекта значительно замедляет прогресс, достигнутый группой разработчиков. Использование сервисов, удаленно управляемых поставщиком облачных услуг, вместо этих компонентов с самостоятельным управлением позволит командам быстро внедрить архитектуру и начать тестирование приложений, которые будут работать в этой среде.

Использование сервисов, предлагаемых поставщиком облачных вычислений, не обязательно усложняет шаблоны архитектуры, однако позволяет мыслить шире и не ограничиваться рутинной работой. Концепция отсутствия локальных ограничений, например ограниченных физических ресурсов, является критическим атрибутом проектирования при создании облачных архитектур, которые позволят системам достичь масштаба, недостижимого другими способами. Вот, например, отдельные области, в которых применение сервисов, управляемых поставщиком облачных услуг, может обеспечить лучше масштабируемую собственную облачную архитектуру:

- использование управляемых балансировщиков нагрузки для разделения компонентов в архитектуре;
- применение управляемых систем хранилищ данных для предоставления только необходимых объемов хранилищ и автоматического масштабирования при импорте большего количества данных;
- использование управляемых РСУБД, которые обеспечивают быструю и эффективную обработку транзакций и при этом изначально обладают устойчивостью и высокой доступностью.

Современные облачные сервисы

Поставщики облачных сервисов развивают свой бизнес и продолжают совершенствовать предложения, позволяя решать с их помощью еще более сложные задачи.

Нынешняя волна инноваций, внедряемых ведущими поставщиками облачных вычислений, предполагает дальнейшее увеличение количества сервисов, при этом поставщик управляет их масштабами и безопасностью и снижает стоимость, сложность и риски для клиента. Один из способов достижения этого — реорганизация существующих технологических платформ таким образом, чтобы они не просто решали конкретные технические проблемы, но и делали это с учетом преимуществ облачных вычислений. Например, в AWS есть сервис управляемых СУБД Amazon Aurora, построенный на полностью распределенном и самовосстанавливающемся хранилище, предназначенном для обеспечения безопасности данных и их распределения по нескольким зонам доступности. Этот сервис повышает полезность предложений управляемых сервисов, специфичных для СУБД, как описано в предыдущем подразделе, позволяя также создавать массив хранилищ, который увеличивается по требованию и настраивается для облака, чтобы обеспечить производительность, в пять раз бóльшую, чем дают аналогичные движки СУБД.

Не все современные удаленно управляемые сервисы представляют собой переработку существующих технологий. С внедрением бессерверных вычислений поставщики облачных технологий убирают недифференцированную рутинную работу не только из процесса эксплуатации, но и из разработки. Поскольку облако может предложить практически безграничные ресурсы, разделение крупных приложений на отдельные функции является следующей большой волной проектирования распределенных систем и ведет непосредственно к формированию архитектур cloud native.

По данным AWS, *«бессерверные вычисления позволяют создавать и запускать приложения и сервисы, не задумываясь о серверах. Бессерверные приложения не требуют от вас подготовки и масштабирования каких-либо серверов и управления ими. Вы можете создавать их практически для любого типа приложения или серверной службы, и все, что требуется для запуска и масштабирования вашего приложения, а также обеспечения высокой доступности, выполняется за вас»*.

Существует много видов бессерверных решений, включая сервисы для вычислений, обработки сообщений и оркестрации, прокси-серверы для API, хранилища, базы данных, средства анализа и инструменты разработчика. Один из ключевых атрибутов, определяющих, является ли облачный сервис бессерверным или только управляемым предложением, — это способ оценки его лицензирования и использования. Бессерверный подход позволяет отказаться от прежней централизованной модели ценообразования, привязанной непосредственно к отдельным серверам, и сделать основной упор на потребление. Время, в течение которого выполняется функция, требуемое количество транзакций в секунду или их комбинация являются общими показателями, которые используются для ценообразования на основе потребления с предложениями без серверов.

Использование усовершенствованных облачных сервисов с собственным управлением и дальнейшее внедрение новых по мере их выпуска показывает высокий уровень зрелости CNMM и позволяет компаниям разрабатывать самые современные

облачные архитектуры. Благодаря данным сервисам и опытной команде разработчиков компания сможет выйти за рамки возможного, когда будут устранены существовавшие ранее ограничения и реализованы действительно безграничные и гибкие возможности. Например, вместо традиционного трехуровневого распределенного вычислительного стека, состоящего из внешнего интерфейса, приложений или промежуточного программного обеспечения и системы оперативной обработки транзакций БД, новый подход к этому шаблону проектирования позволит ввести шлюз API, который использует вычислительный контейнер, управляемый событиями, в качестве конечной точки и управляемую и масштабируемую СУБД NoSQL для хранения. Все эти компоненты могут попасть в бессерверную модель, что дает возможность команде разработчиков сосредоточиться на бизнес-логике, а не на том, как достичь требуемого масштаба.

За бессерверными и другими продвинутыми удаленно управляемыми сервисами будущее. В настоящее время передовые технологии облачных вычислений — это предложения, выпускаемые в областях искусственного интеллекта, машинного и глубокого обучения. У поставщиков развитых облачных сервисов есть сервисы, которые подпадают под эти категории, а постоянные инновации происходят в гипермасштабе. Мы все еще находимся на раннем этапе развития искусственного интеллекта, но команды разработчиков должны думать наперед.

Обзор оси облачных сервисов

В разделе «Ось 1. Облачные сервисы» описаны компоненты, которые могут составить облачную архитектуру, и продемонстрированы современные подходы к созданию приложений и их использованию. Как и в случае с любыми принципами проектирования в CNMM, разработка начинается с понимания основной идеи и продвигается по мере того, как команда разработчиков все лучше овладевает методами проектирования крупномасштабных систем. Однако применение компонентов облачных вычислений — это лишь один из принципов проектирования, необходимых для создания развитой архитектуры cloud native. Они применяются в сочетании с двумя другими принципами — автоматизацией и ориентированностью на приложения для создания систем, которые могут безопасно и надежно использовать преимущества облака.

Ось 2. Проектирование, ориентированное на приложения

Второй принцип облачных технологий заключается в том, как будет спроектировано и разработано само приложение. В этом разделе основное внимание будет

уделено фактическому процессу разработки приложений, а также будут определены приемы, которые позволяют создавать собственные облачные архитектуры. Подобно другим способам проектирования CNMM, разработка приложений cloud native — это постепенный процесс с применением разных подходов, которые обычно сменяют друг друга. В конечном итоге при использовании и других принципов CNMM результатом будет сложная и надежная облачная архитектура с богатыми возможностями, которая способна развиваться по мере совершенствования мира облачных вычислений (рис. 1.4).

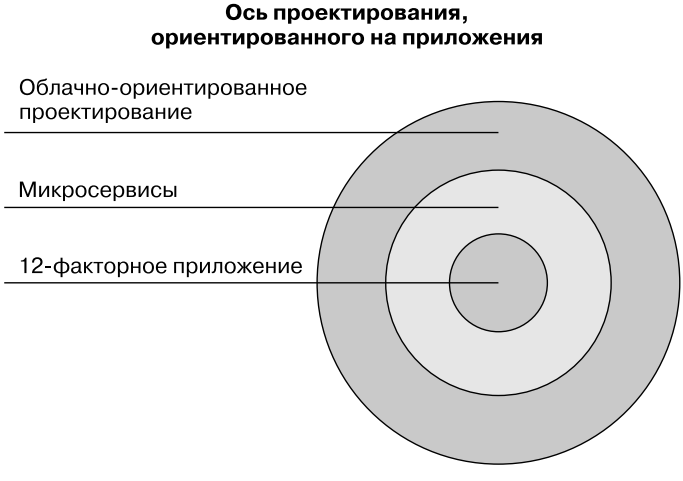


Рис. 1.4

Принципы проектирования 12-факторных приложений

«Двенадцатифакторное приложение» — это методология создания приложений программного обеспечения в виде сервисов, называемых *веб-приложениями* (web apps) или *software-as-a-service* (SaaS) (https://12factor.net/). Эта методология была предложена в конце 2011 года, и ее факторы часто называют базовыми строительными блоками для разработки масштабируемых и надежных приложений cloud native. Ее принципы применяются к приложениям, написанным на любом языке программирования и использующим любую комбинацию вспомогательных сервисов (база данных, очередь, кэш-память и т. д.), они становятся все более полезными на любой платформе облачного поставщика. Идея, лежащая в основе 12-факторного приложения, заключается в том, что при разработке приложений необходимо учитывать 12 важных факторов, которые сводят к минимуму время и затраты на добавление новых разработчиков, обеспечивают возможность чистого взаимодействия со средой, позволяют развернуть приложение у поставщиков

облачных сервисов, способны минимизировать расхождения между средами и разрешить масштабирование приложения. В следующей таблице перечислены эти двенадцать факторов (https://12factor.net/).

№ п/п	Фактор	Описание
1	Кодовая база	Одна кодовая база, отслеживаемая в системе контроля версий, — множество развертываний
2	Зависимости	Явно объявляйте и изолируйте зависимости
3	Конфигурация	Сохраняйте конфигурацию в среде выполнения
4	Вспомогательные сервисы	Считайте вспомогательные сервисы подключаемыми ресурсами
5	Сборка, релиз, выполнение	Строго разделяйте стадии сборки и выполнения
6	Процессы	Запускайте приложение как один или несколько процессов, не сохраняющих внутреннее состояние (stateless)
7	Привязка портов	Экспортируйте сервисы через привязку портов
8	Параллелизм	Масштабируйте приложение с помощью процессов
9	Утилизируемость	Максимизируйте надежность с помощью быстрого запуска и корректного завершения работы
10	Паритет разработки/работы приложения	Делайте среды разработки, промежуточного и рабочего развертывания максимально похожими
11	Ведение журнала	Считайте журнальные записи потоком событий
12	Задачи администрирования	Выполняйте задачи администрирования/управления с помощью разовых процессов

В предыдущих разделах этой главы уже обсуждалось, как CNMM учитывает несколько факторов из этой методологии. Например, *фактор 1* — хранение базы кода в репозитории — лучшая стандартная практика. *Факторы 3*, *10* и *12* основаны на разделении сред, но при этом они не должны отделяться друг от друга с точки зрения кода и конфигурации. *Фактор 5* гарантирует, что у вас есть чистый и повторяемый конвейер CICD с разделением функций. А *фактор 11* касается обработки журналов как потоков событий таким образом, чтобы их можно было анализировать и обрабатывать практически в реальном времени. Остальные факторы хорошо согласуются с разработкой облака по той простой причине, что они сосредоточены на том, чтобы быть автономными *(фактор 2)*, рассматривать все как услугу *(факторы 4* и *7)*, обеспечивать эффективное масштабирование *(факторы 6* и *8)* и корректно обрабатывать неисправности *(фактор 9)*. Разработка приложения с использованием 12-факторной методологии — не единственный способ создания нативных облачных архитектур. Тем не менее он предлагает стандартизированный набор принципов, которые, если их придерживаться, позволят приложению стать более развитым согласно CNMM.

Монолитная, сервис-ориентированная и микросервисная архитектуры

Шаблоны проектирования архитектур постоянно развиваются с применением последних технологических инноваций. *Монолитные архитектуры* были популярными на протяжении длительного времени, во многом благодаря высокой стоимости физических ресурсов и низким темпам. Эти шаблоны хорошо подходят для рабочих компьютеров и мейнфреймов, и даже сегодня существует множество приложений, работающих как монолитные архитектуры. Поскольку технологические процессы и бизнес-требования стали более сложными, а скорость выхода на рынок приобретает все большее значение, для поддержки этих требований были развернуты дополнительные монолитные приложения. В конце концов, эти монолитные приложения должны были взаимодействовать друг с другом, чтобы обмениваться данными или выполнять функции, которые имелись в других системах. Это взаимодействие стало предшественником *ориентированных на сервисы архитектур (SOA)*, которые позволили командам разработчиков создавать меньшие по сравнению с монолитными компоненты приложений, внедрять компоненты промежуточного программного обеспечения для поддержания связи и исключать возможность доступа к компонентам иначе, кроме как через специальные конечные точки. Проекты SOA приобрели все большую популярность во время бума виртуализации, поскольку развертывание сервисов на виртуализированном оборудовании стало проще и дешевле.

Сервис-ориентированные архитектуры состоят из двух или более компонентов, которые предоставляют свои сервисы другим сервисам через специальные протоколы связи. Последние часто называют *веб-сервисами*, они состоят из нескольких общих протоколов (WSDL, SOAP и RESTful HTTP) в дополнение к протоколам обмена сообщениями, таким как JMS. По мере роста сложности различных протоколов и сервисов все шире распространялось использование *сервисной шины предприятия (enterprise service bus, ESB)* в качестве промежуточного уровня. Это позволило сервисам изолировать свои конечные точки, и сервисная шина предприятия могла позаботиться о переводах сообщений из различных источников, чтобы получить правильно отформатированный вызов желаемой системы. Хотя этот подход упростил взаимодействие между сервисами, он в то же время добавил сложности логике промежуточного программного обеспечения, необходимой для преобразования вызовов сервисов и обработки рабочих процессов. Это часто приводило к созданию весьма сложных SOA-приложений, в которых код приложения для каждого из компонентов необходимо было развертывать одновременно, что приводило к «большому взрыву» и рискованному масштабному развертыванию в составном приложении.

Подход SOA положительно повлиял на проблемные последствия непредвиденных изменений, которые изначально имели монолитные архитектуры, путем разделения основных компонентов на отдельные приложения. Однако появилась проблема сложности развертывания. Она проявлялась в том, что вызывала так

много взаимозависимостей, что часто требовалось одно большое развертывание для всех приложений SOA. В результате эти рискованные развертывания — *«большой взрыв»* — часто предпринимались лишь несколько раз в год, а резкое снижение скорости замедляло выполнение бизнес-требований. По мере того как облачные вычисления стали более распространенными, а ограничения локальных сред начали постепенно исчезать, появился новый архитектурный шаблон — *микросервисы*. Действуя через облако, команды разработчиков приложений больше не должны были ждать месяцами, чтобы получить вычислительные возможности для тестирования своего кода, также они не были ограничены физическими ресурсами.

Стиль архитектуры микросервисов имеет ту же распределенную природу, что и SOA, только сервисы делятся на еще более обособленные и слабо связанные прикладные функции. Микросервисы не только уменьшают последствия непредвиденных изменений, даже дополнительно изолируя функции, но и значительно увеличивают скорость развертывания приложений, рассматривая каждую микросервисную функцию как собственный компонент. Деятельность небольшой группы специалистов DevOps, ответственной за конкретный микросервис, позволит осуществлять непрерывную интеграцию и непрерывную доставку кода небольшими порциями, что увеличивает скорость и обеспечивает быстрый откат в случае непредвиденных проблем, возникающих в сервисе.

Микросервисы и облачные вычисления хорошо сочетаются друг с другом, и зачастую микросервисы считаются наиболее развитым типом архитектуры cloud native на данный момент. Причина такой хорошей сочетаемости заключается в том, что поставщики облачных услуг часто разрабатывают их в виде отдельных строительных блоков, которые можно использовать различными способами для достижения бизнес-результата. Подход, основанный на стандартных блоках, дает командам разработчиков приложений возможность комбинировать и сопоставлять сервисы для решения своих проблем, вместо того чтобы применять конкретный тип хранилища данных или язык программирования. Это привело к увеличению числа инноваций и шаблонов проектирования, использующих преимущества облака, таких как бессерверные вычислительные сервисы, скрывающих управление ресурсами от команд разработчиков, что позволяет им сосредоточиться на бизнес-логике.

Особенности облачно-ориентированного проектирования

Независимо от используемой методологии или облачно-ориентированного шаблона проектирования, который в итоге реализуется, существуют определенные характеристики, которые должны быть общими для всех облачно-ориентированных архитектур. Не все из этих характеристик необходимо рассматривать в контексте облачных архитектур, однако их реализация делает систему более развитой, поэтому они находятся на более высоком уровне CNMM. К таким характеристикам

относятся инструментарий, безопасность, распараллеливание, отказоустойчивость, управляемость событиями и забота о будущем.

- *Инструментарий*. Включение инструментария приложения — это больше чем просто анализ потоков журналов. Для этого требуется способность контролировать и измерять производительность приложения в режиме реального времени. Добавление инструментария напрямую приведет к тому, что приложение сможет распознавать условия задержки, сбои компонентов из-за сбоев системы и другие характеристики, важные для конкретного бизнес-приложения. Инструментарий имеет решающее значение для многих других соглашений разработки, поэтому включение его в качестве объекта первого класса в приложение принесет выгоду в долгосрочной перспективе.
- *Безопасность*. Все приложения нуждаются во встроенной защите. Тем не менее проектирование архитектуры безопасности в cloud native имеет решающее значение для обеспечения использования приложением в своих интересах сервисов безопасности облачных поставщиков, а также сторонних сервисов безопасности на всех уровнях. Это позволит повысить защиту приложения и уменьшить степень поражения в случае атаки или взлома.
- *Распараллеливание*. Производительность приложения в ходе масштабирования зависит непосредственно от его способности выполнять отдельные процессы параллельно. Это предусматривает возможность параллельного выполнения одного и того же набора функций или одновременное выполнение множества различных функций в приложении.
- *Отказоустойчивость*. Важно учитывать, как приложение будет обрабатывать неисправности и в каком объеме при этом продолжать работать. Использование инноваций поставщиков облачных услуг, таких как развертывание в нескольких физических центрах обработки данных, применение нескольких разделенных уровней приложения, а также автоматизация запуска, завершения работы и миграции компонентов приложения между местоположениями поставщиков — все это способы обеспечения отказоустойчивости приложения.
- *Управляемость событиями*. Приложения, управляемые событиями, могут использовать методы, которые анализируют события для выполнения действий, будь то бизнес-логика, изменение отказоустойчивости, оценка безопасности или автоматическое масштабирование компонентов приложения. Все события регистрируются и анализируются с помощью передовых методов машинного обучения, что позволяет применять дополнительную автоматизацию при выявлении новых событий.
- *Забота о будущем*. Размышления о будущем — это важнейший способ гарантировать, что приложение будет развиваться в соответствии с CNMM с течением времени и появлением инноваций. Реализация этих соображений поможет ему стать перспективным. Однако все приложения должны быть оптимизированы с помощью автоматизации и постоянных улучшений кода, чтобы всегда иметь возможность обеспечить требуемые бизнес-результаты.

Обзор оси проектирования, ориентированного на приложения

Существует много различных методологий, которые можно использовать для создания облачного приложения, включая микросервисы, шаблоны проектирования 12-факторных приложений и соглашения облачного проектирования. Не существует единого правильного пути разработки облачных приложений, и, как и любые фрагменты CNMM, они будут развиваться по мере появления новых соглашений. Приложение достигнет максимального уровня развития для данной оси, как только будут реализованы большинство этих принципов проектирования.

Ось 3. Автоматизация

Третий и последний принцип cloud native — это автоматизация. В этой главе подробно обсуждались и объяснялись другие принципы CNMM, в частности, почему использование облачных сервисов и проектирование, ориентированное на приложения, позволяют архитектурам cloud native становиться более масштабными. Однако сами по себе они не дают возможности системе действительно использовать все преимущества облака. Если бы система была разработана с использованием самых современных сервисов, но эксплуатация приложения выполнялась вручную, было бы трудно понять, для чего она предназначена. Этот тип автоматизации часто называют *«инфраструктура как код» (Infrastructure as Code, IaC)*, и он эволюционирует, позволяя создавать сложные архитектуры cloud native. Поставщики облачных услуг обычно разрабатывают все свои сервисы так, чтобы они были конечными точками API, что позволяет программным вызовам создавать, изменять или уничтожать сервисы. Этот подход — движущая сила IaC. Для сравнения, раньше команда эксплуатации отвечала не только за проектирование и конфигурирование инфраструктуры, но и за физическую настройку и развертывание компонентов.

Благодаря автоматизации IaC команды эксплуатации теперь могут сосредоточиться на разработке приложений и полагаться на поставщика облачных услуг в том, чтобы справиться с недифференцированной рутинной работой при распределении ресурсов. Инфраструктура как код затем обрабатывается, как любой другой артефакт развертывания для приложения, хранится в репозиториях исходного кода и поддерживается для обеспечения долгосрочной согласованности наращивания среды. Степень автоматизации по-прежнему высока, причем на ранних этапах основное внимание уделялось созданию среды, настройке ресурсов и развертыванию приложений. По мере развития решения автоматизация будет совершенствоваться, включая в себя более развернутый мониторинг, масштабирование и повышение производительности, и в конечном итоге будет предусматривать аудит, соблюде-

ние нормативных требований, управление и оптимизацию всего решения. На следующем этапе при автоматизации самых передовых архитектур используются искусственный интеллект, а также методы машинного и глубокого обучения для самовосстановления и самостоятельного управления системой с целью изменения ее структуры в зависимости от текущих условий.

Автоматизация — это ключ к обеспечению масштабирования и безопасности, необходимых для облачной архитектуры (рис. 1.5).

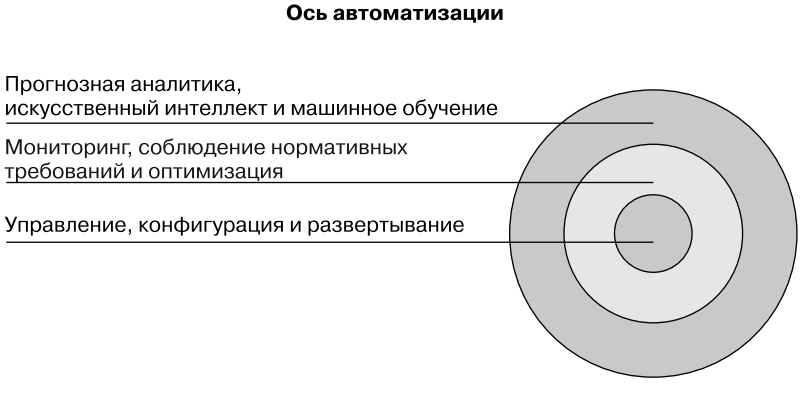


Рис. 1.5

Управление средой, конфигурация и развертывание

Проектирование и развертывание приложения в облаке, а также управление им сложны, так как все системы должны быть настроены и сконфигурированы. Эти процессы можно упростить и согласовать путем реализации процессов создания и настройки среды в виде кода. Существует множество облачных сервисов и ресурсов, которые выходят далеко за рамки традиционных серверов, подсетей и физического оборудования, управляемых локально. На этом этапе оси автоматизации основное внимание уделяется предоставлению среды на основе API, конфигурации системы и развертыванию приложений, что позволяет клиентам использовать код для выполнения этих повторяющихся задач.

Независимо от того, является ли решение крупной и сложной реализацией для корпоративной компании или относительно простым развертыванием системы, применение автоматизации для управления согласованностью имеет решающее значение при создании архитектуры cloud native. Для больших и сложных решений или в тех случаях, когда нормативные требования включают необходимость

разделения обязанностей, компании могут использовать подход, называемый *«инфраструктура как код»* (IaC), чтобы разделить разные команды эксплуатации и сосредоточить их внимание только на определенной области, например на базовой инфраструктуре, создании сетей, безопасности и мониторинге. В других случаях все компоненты может обрабатывать одна команда, возможно, даже команда разработчиков, если используется модель DevOps. Независимо от того, как развивается IaC, важно в ходе этого процесса обеспечить гибкость и согласованность, частое развертывание систем и точное соблюдение требований проектирования.

Существует множество способов управления системой с помощью IaC. В некоторых случаях каждый раз, когда происходит изменение среды, полная автоматизация IaC реализуется для замены существующей среды. Это называется *неизменяемой инфраструктурой*, поскольку системные компоненты никогда не обновляются, а каждый раз заменяются новой версией или конфигурацией. Это позволяет компании уменьшить влияние среды или конфигурации и обеспечивает строгий способ подтверждения корректности процессов администрирования и уменьшения проблем безопасности, которые могут появиться при реализации вручную.

Несмотря на то что подход с неизменяемой инфраструктурой имеет свои преимущества, может оказаться нереально каждый раз заменять среду полностью, поэтому необходимо вносить изменения на уровне конкретных компонентов. При таком подходе автоматизация все еще имеет решающее значение для обеспечения согласованной реализации, однако это сделает облачные ресурсы изменяемыми или даст им возможность меняться со временем. Существует множество поставщиков, у которых есть продукты для обеспечения автоматизации на уровне инстанса, и большинство поставщиков облачных услуг имеют управляемые предложения для обеспечения этого типа автоматизации. Эти инструменты позволят запускать код или сценарии в среде для внесения изменений. Такие сценарии будут частью инфраструктуры как артефакты развертывания кода и станут разрабатываться и поддерживаться так же, как набор неизменяемых сценариев.

Управление средой и ее конфигурация — не единственный способ автоматизации на базовом уровне. Развертывание кода и гибкость также очень важны для обеспечения полностью автоматизированной архитектуры cloud native. На рынке существует множество инструментов, позволяющих автоматизировать весь конвейер развертывания, часто называемый *непрерывной интеграцией, непрерывным развертыванием (CICD)*. Конвейер развертывания кода часто включает в себя все аспекты процесса, от регистрации кода, автоматической компиляции с анализом кода, упаковки и развертывания до применения конкретных сред для обеспечения чистого развертывания. Используемые по принципу «инфраструктура как код», конвейеры CICD обеспечивают архитектуре cloud native исключительную гибкость и согласованность.

Мониторинг, соблюдение нормативных требований и оптимизация с помощью автоматизации

Архитектуры cloud native, которые используют сложные сервисы и охватывают несколько географических регионов, требуют возможности часто меняться в зависимости от шаблонов использования и других факторов. Применение автоматизации для мониторинга всего диапазона решения, обеспечения соответствия стандартам компании или нормативным требованиям, а также постоянной оптимизации использования ресурсов демонстрирует растущий уровень зрелости. Как и при любой эволюции, построение на предыдущих этапах зрелости дает возможность задействовать передовые методы, которые позволяют повысить масштабность решения.

Одними из наиболее важных сведений, которые можно собрать, являются данные мониторинга, которые поставщики облачных технологий будут включать в свои предложения. У серьезных поставщиков облачных услуг сервисы мониторинга встроены в их сервисы, они способны собирать показатели, события и журналы этих сервисов, которые в противном случае были бы недоступны. Использование таких сервисов мониторинга для запуска основных событий — это тип автоматизации, который обеспечивает общее состояние системы. Например, система, задействующая парк вычислительных виртуальных машин в качестве логического уровня компонента сервиса, обычно ожидает определенного количества запросов, но периодически всплеск числа запросов приводит к быстрому увеличению нагрузки на ЦП и росту сетевого трафика в этих инстансах. При правильной настройке сервис облачного мониторинга обнаружит это увеличение и запустит дополнительные инстансы, чтобы выровнять нагрузку до более приемлемых уровней и обеспечить надлежащую производительность системы. Запуск дополнительных ресурсов обусловлен проектной конфигурацией системы, для которой требуется автоматизация IaC, чтобы гарантировать, что новые инстансы будут развернуты с использованием таких же точно конфигурации и кода, что и все остальные в кластере. Этот тип деятельности часто называют *автоматическим масштабированием*, он действует и в обратном порядке, удаляя инстансы после того, как число запросов уменьшилось.

Автоматизация соответствия конфигураций среды и системы становится все более важной для крупных корпоративных клиентов. Включение автоматизации в постоянные аудиторские проверки соответствия между компонентами системы показывает высокий уровень зрелости по оси автоматизации. Сервисы для получения моментальных снимков конфигурации позволяют создать подробную картину структуры среды и хранятся в виде текста, что дает возможность долгосрочного анализа. Используя автоматизацию, такие снимки можно сравнить с предыдущими представлениями о среде, чтобы убедиться, что конфигурация не изменилась. Вдобавок текущий снимок можно сравнить с требующимися и совместимыми конфигурациями, которые будут соответствовать требованиям аудита в отраслях, регулируемых законодательством.

Оптимизация облачных ресурсов — это область, которую легко проигнорировать. Система была спроектирована до появления облака, и оценивалась емкость, необходимая для работы этой системы в пиковых условиях. Поэтому еще до того, как система была создана, было куплено дорогое и сложное аппаратное и программное обеспечение. Из-за этого часто случалось, что значительная часть избыточной емкости простаивала в ожидании увеличения количества запросов. При использовании облака эта проблема почти исчезает, однако разработчики системы все еще сталкиваются с ситуациями, когда неизвестно, какая емкость потребуется. Для решения этой проблемы можно использовать автоматическую оптимизацию, чтобы постоянно проверять все компоненты системы и, оценивая исторические тренды, понимать, чрезмерна или недостаточна нагрузка на эти ресурсы. Можно добиться этого с помощью автоматического масштабирования, однако есть намного более сложные способы, которые, будучи правильно реализованными, обеспечат дополнительную оптимизацию. Например, автоматизированная проверка запущенных инстансов во всех средах на предмет неиспользуемой емкости и их отключение или аналогичная проверка для выключения всех сред разработки по ночам и выходным могут сэкономить компании много денег.

Одним из ключевых способов получения развитой облачной архитектуры с точки зрения мониторинга, соблюдения нормативных требований и оптимизации является использование обширной инфраструктуры ведения журналов. Загрузка данных в эту среду и анализ этих данных для принятия решений — сложная задача, и проектная группа должна хорошо понимать, как работают различные компоненты, и обеспечить постоянный сбор всех необходимых данных. Такого рода инфраструктура помогает избавиться от мысли, что журналы — это файлы, которые нужно собирать и хранить, и начать думать о них как о потоках событий, которые следует анализировать в реальном времени на предмет каких-либо аномалий. Например, довольно быстрым способом реализации каркаса журналирования было бы использование ElastiCache, Logstash и Kibana, часто называемого стеком ELK, для захвата всех типов событий системного журнала, событий журнала сервисов облачных поставщиков, а также других журналов событий.

Прогнозная аналитика, искусственный интеллект, машинное обучение и не только

По мере того как система развивается и становится все более глубоко автоматизированной, она все больше полагается на данные, которые сама генерирует для анализа и обработки. Подобно проекту мониторинга, соответствия и оптимизации, рассмотренному ранее, развитая облачная архитектура будет постоянно анализировать потоки событий журнала для выявления аномалий и неэффективности, однако признаком наибольшей зрелости становится использование искусственного интеллекта (AI) и машинного обучения (ML) для прогнозирования того, как события могут повлиять на систему, и внесения корректировок, предотвращающих снижение производительности, безопасности или других показателей эффектив-

ности бизнеса. Чем обширнее круг разнородных источников, из которых поступают данные о событиях, и чем дольше они сохраняются, тем больше исходных данных для принятия мер будут получать эти инструменты.

Используя строительные блоки автоматизации, обсуждавшиеся ранее, в сочетании с AI и ML, система получает множество возможностей решить проблему, которая может повлиять на бизнес.

Данные важны, когда дело доходит до прогнозной аналитики и машинного обучения. Бесконечный процесс обучения системы классификации событий требует времени, данных и автоматизации. Основой методов AI и ML является возможность сформировать прогноз с ориентиром на корреляцию, казалось бы, не связанных между собой событий и данных. На основании этого прогноза подготавливается ряд действий, которые ранее приводили к коррекции аналогичных аномалий. Автоматическое реагирование на событие, которое соответствует гипотезе о причине аномалии, и реализация корректирующих действий — это пример использования прогнозирующей аналитики на основе ML для решения проблемы до того, как она станет влиять на бизнес. Кроме того, всегда будут возникать ситуации, когда регистрируется новое событие и невозможно точно соотнести его с ранее известной аномалией, основываясь на ранее полученных данных. Даже несмотря на это, отсутствие корреляции само по себе показательно и позволяет перекрестно связывать события, аномалии и ответы и тем самым накапливать опыт.

Существует много примеров того, как использование ML на наборах данных показывает корреляцию, которую не может увидеть человек, просматривающий те же самые наборы данных, например, как часто неудачная попытка пользователя войти в систему приводила к блокировке по сравнению с миллионами других попыток и были ли эти блокировки результатом безобидного ввода неверного пароля из-за забывчивости или грубой атаки с целью получить доступ в систему. Поскольку алгоритм может искать все необходимые наборы данных и коррелировать результаты, он сможет идентифицировать закономерности, говорящие о том, безопасное событие или злонамеренное. Используя выходные данные этих шаблонов, можно предпринять превентивные действия для предотвращения возможных проблем с безопасностью, быстро изолируя ресурсы веб-интерфейса или блокируя запросы от пользователей, которые считаются вредоносными из-за того, откуда они поступают (IP-адрес или конкретная страна), типа передаваемого трафика (DDoS-атака) или другого сценария.

Этот тип автоматизации, если он будет правильно реализован в системе, позволит создать самые современные из возможных на сегодня архитектур. Учитывая текущее состояние доступных облачных сервисов, использование прогнозной аналитики, искусственного интеллекта и машинного обучения — это самый передовой способ создания продвинутой архитектуры cloud native. Однако по мере того, как сервисы станут более совершенными, появятся дополнительные методы, и разработчики будут продолжать использовать их, чтобы гарантировать надежность своих систем и предотвратить ущерб для бизнеса.

Обзор оси автоматизации

Автоматизация предоставляет широкие возможности использования архитектуры cloud native. Уровень зрелости автоматизации будет расти от простой настройки сред и компонентов до выполнения расширенного мониторинга, обеспечения соответствия и оптимизации решения в целом. В сочетании с увеличением количества инноваций в предложениях поставщиков облачных услуг усовершенствованная автоматизация и использование искусственного интеллекта и машинного обучения позволят предпринять превентивные действия для устранения распространенных, известных и все чаще неизвестных аномалий в системе. Внедрение и автоматизация услуг облачных поставщиков составляют два из трех важнейших принципов проектирования для CNMM. Третьим является разработка архитектуры приложения.

Переход в облако

Большие и маленькие, новые и опытные компании видят преимущества облачных вычислений. Существует множество способов начать использовать облако, и часто все зависит лишь от готовности высшего руководства принять необходимые изменения. Независимо от типа организации переход к облачным вычислениям — это путешествие, которое потребует времени, усилий и настойчивости для достижения успеха. Руководству компании легко сказать, что они хотят перейти к использованию облака, однако для большинства переход — сложная процедура. Состоявшимся организациям, которые работают на базе программного обеспечения старых версий, а также управляют центрами обработки данных, придется не только определять дорожную карту и планировать миграцию, но и управлять людьми и процессами. Путь более молодых компаний, не имеющих большого технического багажа в виде традиционных рабочих процессов, окажется короче, и облачные вычисления станут местом ознакомительных экспериментов. Тем не менее переход к предприятию cloud native все равно займет время.

Решение «облако превыше всего»

Облачные вычисления здесь первостепенны. Много лет назад было много дискуссий о том, стоит ли компаниям внедрять облачно-ориентированную модель в погоне за самыми новыми и лучшими технологиями. Однако на данный момент почти каждая компания сделала первый шаг к облачным вычислениям и многие приняли решение стать облачно-ориентированной организацией. На самом базовом уровне принятие этого решения просто означает, что все новые рабочие нагрузки будут развернуты в пространстве, предоставленном выбранным поставщиком облачных услуг, если не будет доказано, что этого недостаточно для удовлетворения бизнес-требований. Иногда это происходит из-за необходимости обеспечивать

информационную безопасность (то есть из-за государственных или нормативных условий), а иногда из-за конкретной технической проблемы или ограничения, характерного для поставщика облачных вычислений, которые трудно преодолеть за короткое время. Несмотря на это, подавляющее большинство новых проектов окажется в облаке различной степени зрелости, как описано ранее в CNMM.

Несмотря на то что это решение широко распространено в современной ИТ-среде, все еще существуют проблемы, которые необходимо решить, чтобы оно стало успешным. ИТ-лидерам и лидерам бизнеса необходимо обеспечить адаптацию их сотрудников и процессов к модели «облако превыше всего». Кроме того, разработка DevOps и гибкой методологии поможет организации преодолеть низкие темпы развития и недостаточную гибкость, присущие каскадным проектам, за счет разделения команд разработки и эксплуатации.

Люди и процессы в облаке меняются

В организациях с крупными ИТ-отделами или долго сотрудничающих с внешними подрядчиками, естественно, будет персонал, владеющий технологиями, которые использовались в компании до этого момента. Переход на любую новую технологию, особенно облачные вычисления, потребует значительного переоснащения, смены персонала и изменения образа мышления сотрудников. Организации могут преодолеть проблемы этих людей, разделив свой ИТ-персонал на два отдельных коллектива: тех, кто поддерживает традиционные рабочие нагрузки и сохраняет оригинальные методологии, и тех, кто работает по модели «облако превыше всего» и внедряет новые технологии и процессы для достижения успеха. Этот подход может работать некоторое время, однако по мере перемещения рабочих нагрузок на целевую облачную платформу все больше и больше людей будут переходить на новую операционную модель. Преимущества этого подхода позволяют избранным сотрудникам, увлеченным и стремящимся к изучению новых технологий и методов, стать первопроходцами, в то время как остальные могут адаптировать свои навыки более систематично.

Одной из специфических областей, которую опытным ИТ-специалистам часто трудно осознать, особенно если они получили опыт развертывания центров обработки данных и освоили много устаревших рабочих процессов, является концепция неограниченных ресурсов. Поскольку большинство поставщиков облачных решений предоставляют практически неограниченные ресурсы, устранение этого ограничения на разработку приложений откроет множество уникальных инновационных способов решения проблем, которые раньше были невозможны. Например, если выполнение пакетного задания зависит от определенного набора процессоров, разработчики часто будут отказываться от распараллеливания процессов, в то время как при неограниченном количестве ЦП для всех процессов может быть предусмотрена возможность параллельного выполнения, что потенциально быстрее и дешевле, чем при проведении множества последовательных операций. Люди,

которые могут мыслить масштабно и обходить ограничения, должны составлять команду первопроходцев.

Процессы также являются серьезным камнем преткновения для организации, ориентированной на облачные вычисления. Многие компании, которые переходят в облако, переходят также от фазы SOA к микросервисам. Следовательно, обычные процессы поддерживают архитектуру и развертывание SOA, которые, скорее всего, замедляют работу и гарантируют, что развертывание по принципу «большого взрыва» в составном приложении будет выполнено правильно и с тщательным тестированием. При реализации подхода «облако превыше всего» и использовании микросервисов цель состоит в том, чтобы развертывать код как можно быстрее и чаще, соответствуя быстро меняющимся бизнес-требованиям. Поэтому изменение процессов для поддержки этой гибкости имеет решающее значение. Например, если организация строго следует ITIL, ей может потребоваться строгая цепочка утверждений с системой сдержек и противовесов, прежде чем любые изменения или развертывание кода можно будет реализовать на производстве. Этот процесс, вероятно, имеет место из-за сложной взаимосвязанной природы составных приложений, и одно незначительное изменение может повлиять на весь набор систем. Однако архитектура микросервисов такова, что поскольку они полностью автономны и публикуют API (обычно), то, пока API не меняется, сам код не будет влиять на другие сервисы. Изменение процессов для множества небольших развертываний или откатов обеспечит скорость и гибкость бизнеса.

Гибкость и DevOps

Облако — это не волшебное место, где нет проблем. Это место, где исчезают некоторые привычные проблемы, но возникают новые. Существующие корпорации уже некоторое время переходят от каскадных методов управления проектами к гибким. Это хорошая новость для компании, которая намеревается стать облачной платформой, поскольку итерации, быстрые сбои и инновации имеют решающее значение для достижения долгосрочного успеха, а гибкие проекты позволяют реализовывать проекты такого типа. По большей части причина, по которой эта методология пользуется популярностью у компаний cloud native, заключается в быстром темпе внедрения инноваций поставщиками облачных услуг. Например, в 2017 году AWS запустил 1430 новых сервисов и функций, то есть почти четыре в день, а в 2018-м их количество вновь увеличилось. При таком количестве инноваций облачные сервисы меняются, и использование гибкой методологии для управления проектами cloud native позволяет компаниям применять их по мере выхода.

DevOps (или объединение команд разработчиков и команд эксплуатации) — это новая операционная модель ИТ, которая помогает преодолеть разрыв между тем, как разрабатывается код и как он работает после развертывания на производстве. Создание единой команды, ответственной за логику кода, тестирование, артефакты

развертывания и эксплуатацию системы, гарантирует, что ничего не будет потеряно в жизненном цикле процесса разработки кода. Данная модель хорошо сочетается с облаком и микросервисами, поскольку позволяет небольшой команде заниматься сервисом в целом, писать любой подходящий код, выполнять развертывание на облачной платформе выбранной компании, а затем управлять этим приложением и занимать наилучшую позицию для решения любых проблем, которые могут возникнуть у приложения после его запуска.

Вместе гибкие методологии и DevOps несут критически важные изменения, необходимые для компаний, которые рассматривают возможность превращения в организацию cloud native.

Облачная операционная среда

Переход в облако методом проб и ошибок займет много времени. Как правило, компания определяет основного поставщика облачных услуг, удовлетворяющего ее требованиям, а в некоторых случаях у нее будет и второй поставщик для реализации конкретных целей. Кроме того, почти все компании начинают с применения гибридной архитектуры, которая позволяет использовать существующие инвестиции и приложения, перемещая рабочие нагрузки в выбранное облако. Часто переход в облако начинается с переноса в него отдельной рабочей нагрузки или ее разработки для облака, что дает команде разработчиков критически важный опыт и помогает создать операционную основу, которую организация будет использовать для облака.

Операционная основа облака

Облако представляет собой обширный набор ресурсов, которые можно использовать для решения всех видов бизнес-задач. Одновременно это и сложный набор технологий, который требует наличия не только профессионалов, способных с ним работать, но и строгой операционной основы, чтобы гарантировать, что это делается безопасно, с учетом стоимости и масштаба. Еще до того, как в облаке будет развернута рабочая нагрузка, компании важно полностью определиться с ее основными составляющими. Нужно предусмотреть все, от структур учетных записей, проектирования виртуальных сетей, региональных/географических требований, структуры системы безопасности с точки зрения таких областей, как управление идентификационными данными и доступом и их соответствие требованиям, до особенностей управления конкретными сервисами, которые будут применяться для различных типов рабочих нагрузок. Понимание того, как использовать IaC (об этом говорилось ранее, при рассмотрении оси автоматизации), также является критическим элементом, который следует определить на ранней стадии.

Как только все решения будут приняты и основа облачной работы подготовлена, начнется работа над первым проектом. В промежутке между принятием решений и развертыванием первых нескольких проектов команды DevOps получат большой опыт быстрой работы, выбора целевой платформы поставщика облачных вычислений, а также определения руководящих принципов и подходов компании к среде cloud native.

Гибридное облако

Компания должна принять решение не только о создании облачной платформы, но и о том, как использовать существующие активы. И если польза облачных вычислений больше не вызывает сомнений, то обсуждение темпа миграции и скорости амортизации существующих активов продолжается. Использование гибридного подхода в начале перехода в облако широко распространено и позволяет компании легко работать со всеми своими специалистами: командой, занятой традиционными процессами, и командой «облако превыше всего». Этот подход обеспечит и более экономный путь к успеху, поскольку не требует взрывного перехода из существующих центров обработки данных в облако, но позволяет отдельным проектам, подразделениям или другим компонентам перемещаться быстрее, чем другие.

Все поставщики облачных услуг предоставляют возможность гибридной архитектуры, которую можно использовать, когда компания хочет одни рабочие нагрузки сохранить в своих центрах обработки данных, а другие перевести в облако. Этот подход обычно включает в себя настройку определенного вида сетевого подключения одного или нескольких центров обработки данных к одному или нескольким географическим регионам поставщика облачных услуг. Такое сетевое подключение может осуществляться, к примеру, в форме VPN по Интернету или различным выделенным линиям (оптоволокно). Независимо от типа подключения результатом должна быть единая сеть, которая делает все ресурсы компании и рабочие нагрузки видимыми друг для друга (с соблюдением ограничений безопасности и управления). Типичные шаблоны для гибридной облачной архитектуры:

- существующие рабочие нагрузки — локально, новые проекты — в облаке;
- производственные нагрузки — локально, непроизводственные — в облаке;
- среда аварийного восстановления — в облаке;
- хранение или архивирование — в облаке;
- локальные рабочие нагрузки — вклиниваются в облако для получения дополнительной емкости.

Со временем, когда все бóльшая рабочая нагрузка переносится в облако или удаляется из локальных сред, центр тяжести смещается в облако и у организации в облаке окажется больше ресурсов, чем в локальных хранилищах. Этот прогресс

естественен и будет означать переломный момент в деятельности компании, которая хорошо знакома с переходом в облако. В конечном счете все рабочие нагрузки компании cloud native будут находиться в облаке и не останется практически никаких вариантов гибридных подключений, поскольку они больше не используются. На этом этапе организация будет считаться продвинутой облачно-ориентированной компанией.

Многооблачный подход

Корпоративным компаниям необходимо распределять свои риски таким образом, чтобы уменьшалась степень поражения в случае возникновения проблемы, будь то стихийное бедствие или нарушение безопасности, либо просто следить за тем, чтобы во всех регионах, где эти компании представлены, клиенты могли пользоваться их услугами. Так что привлекательность многооблачной среды велика, и некоторые крупные организации выбирают такой путь, переходя в облако. В подходящих условиях этот подход имеет смысл и дает дополнительную уверенность в том, что бизнес может противостоять конкретным типам проблем. Тем не менее для большинства компаний данный тип архитектуры значительно увеличит сложность внедрения облака, а возможно, и замедлит его.

Системные интеграторы, для которых управление сложностью и изменениями является основой профессии, часто распространяют мифы о многооблачных архитектурах и процессах развертывания. Они хотят продвигать наиболее сложную архитектуру из всех возможных, чтобы компания постоянно нуждалась в их услугах для обеспечения бесперебойной работы ИТ-подразделений. Использование *нескольких облаков* — это крайний способ решения данной проблемы, поскольку для этого потребуется вдвое больше знаний из области облачных вычислений, и вдвое больше гибридных или межоблачных соединений. Зачастую облачный брокер обещает, что одна платформа сможет управлять ресурсами в нескольких облаках и локально, чтобы упростить облачные операции. Проблема этого подхода в том, что в действительности посредники сводят широкие возможности облачных поставщиков к наименьшему общему знаменателю, ограничиваясь, как правило, лишь инстансами, хранилищами, балансировщиками нагрузки и т. д.; они не в состоянии предоставить доступ к самым инновационным услугам того или иного облачного поставщика. Все это душит препятствует развитию облачной архитектуры и вынуждает компании задействовать операционную модель, аналогичную той, которую они использовали до перехода в облако. При этом часто приходится платить другой компании за управление своими средами, не получая большой пользы от работы в облаке.

Другой распространенный способ использования нескольких облаков — применение контейнеров для перемещения рабочих нагрузок между облаками. Теоретически этот подход позволяет решить множество проблем, которые порождает мультиоблачный код. В настоящее время существует множество инноваций

с использованием этого подхода, но способность успешно перемещать контейнеры между облаками все еще находится в зачаточном состоянии. По мере появления дополнительных платформ, инструментов и совершенствования процесса наверняка появится новый способ создания передовых архитектур cloud native.

Компании, которые находятся в фазе перехода в облако и рассматривают многооблачный подход, должны спросить себя, зачем им это. Мы считаем, что организации получат бóльшую скорость и эффективность на ранних и средних этапах своего пути, если выберут одного поставщика облачных услуг и сосредоточат на нем все усилия по переоснащению, а не попытаются добавить второе облако в архитектуру. Так что выберите путь, который будет наилучшим образом соответствовать потребностям вашего бизнеса и корпоративной культуре организации.

Миграция приложений в масштабе

Сначала компания принимает решение стать организацией, ориентированной на облачные вычисления, и создает команду DevOps, затем выбирает поставщика облачных услуг и создает целевой фонд для облачных операций. После того как это будет сделано, придет время расширения и наращивания миграции. Цель компании cloud native состоит в том, чтобы сократить свои центры обработки данных и рабочие процессы, которыми она управляет сама, и максимально перенести их в облако. Существует три основных пути:

- миграция методом lift-and-shift устаревших рабочих процессов в облачную среду;
- реорганизация устаревших рабочих процессов для оптимизации в облачной среде;
- разработка архитектуры cloud native с чистого листа.

В большинстве крупных корпораций все три варианта будут реализованы в разных частях устаревших рабочих процессов. В небольших компаниях может использоваться комбинация любых из них в зависимости от желаемых результатов.

Миграция методом lift-and-shift

Миграция по принципу lift-and-shift (дословно «поднять и перенести») — это процесс перемещения существующих рабочих нагрузок в уже реализованную целевую облачную операционную среду. Он обычно выполняется в отношении приложений, сгруппированных по бизнес-подразделениям, технологическому стеку или уровню сложности какого-то другого типа показателей. Миграция методом lift-and-shift в ее самой чистой форме — это буквально создание побитовых копий существующих инстансов, баз данных, хранилища и т. д., которое в реальности выполняется редко, поскольку рентабельность такого решения в облачной среде будет мизерной.

Например, перемещение 100 инстансов из локального центра обработки данных в облако без изменения размера или учета параметров масштабирования, скорее всего, увеличит их стоимость для компании.

Более распространенный вариант метода lift-and-shift — это *миграция lift-tinker-shift*, когда перемещается большинство рабочих процессов, однако определенные компоненты обновляются или заменяются на специфичные для облачных сервисов. Например, перемещение 100 инстансов из локального центра обработки данных в облако со стандартизацией для конкретной операционной системы (скажем, Red Hat Enterprise Edition), перемещение всех баз данных в управляемый сервис облачного поставщика (например, Amazon Relational Database Service) и хранение резервных или архивных файлов в хранилище BLOB-объектов (например, Amazon Simple Storage Service) — это миграция lift-tinker-shift. Данный тип миграции, скорее всего, позволит сэкономить компании много денег, воспользоваться некоторыми наиболее продвинутыми облачными услугами и обеспечить значительные долгосрочные преимущества при будущих развертываниях.

Реорганизация миграции

Компании, которые действительно превращаются в организацию cloud native, скорее всего, предпочтут реорганизовать большую часть своих прежних рабочих процессов, чтобы воспользоваться преимуществами масштабирования и инноваций, которые может предложить облако. Рабочие нагрузки, выбранные для переноса в облако и перестроенные в процессе, могут потребовать больше времени для перемещения, но после завершения они будут соответствовать какому-то аспекту модели CNMM и считаться облачно-ориентированными. Такого рода миграция не требует переписывания проектов с нуля, но она также не позволяет взять готовый проект и перенести его целиком в облако. Она предназначена для того, чтобы значительная часть рабочих процессов приложения была переписана или переформатирована, поэтому соответствуют собственным облачным стандартам. Например, в составном приложении имеется 100 инстансов, использующих традиционную архитектуру SOA, содержащую пять различных рабочих процессов с ESB для передачи трафика. Чтобы реорганизовать это составное приложение, компания решит избавиться от ESB, разбить отдельные рабочие процессы на более функциональные микросервисы, удалить как можно больше инстансов, используя бессерверные облачные сервисы, и переформатировать базу данных из реляционной в NoSQL.

Миграция рабочих процессов с применением реорганизации — это отличный способ для разработчиков команды DevOps создать качественный продукт, углубиться в проектирование архитектуры и использовать все новые навыки и методы для перехода в cloud native. Мы считаем, что в дальнейшем большинство миграционных проектов будут реорганизовывать существующие рабочие нагрузки, чтобы использовать преимущества облачных вычислений.

Компании cloud native

Хотя технически это не миграция, компании cloud native, которые создают новые приложения, предпочтут пройти весь цикл разработки с учетом нативной облачной архитектуры. Даже перераспределенные рабочие процессы по какой-либо причине не смогут полностью изменить свои базовые технологии. Когда компания решает полностью переориентироваться на облако, она избавляется от всех устаревших подходов к разработке, ограничений масштабирования, медленного развертывания и работников с устаревшими навыками, применяя лишь самые новые и лучшие облачные сервисы, архитектуры и методы. Компании, которые дошли до этого этапа, становятся действительно cloud native и настраиваются на долгосрочный успех в разработке и развертывании бизнес-приложений.

Пример использования архитектуры cloud native: Netflix

Когда речь заходит об облачных компаниях со своим видением будущего, в пример часто приводят Netflix. Но почему? В этом разделе мы разберем действия, которые она предприняла, чтобы достичь современного уровня. Используя CNMM, будем обсуждать каждую из осей и примем во внимание ключевые моменты, чтобы продемонстрировать их зрелость в контексте перехода на облачно-ориентированные технологии.

Переход

Как и любые крупные миграции в облако, переход Netflix не произошел за одну ночь. Еще в мае 2010 года компания Netflix открыто рекламировала AWS в качестве избранного партнера по облачным вычислениям. Следующая цитата взята из пресс-релиза, который обе компании тогда опубликовали (http://phx.corporate-ir.net/phoenix.zhtml?c=176060p=irol-newsArticleID=1423977): *«Amazon Web Services сегодня объявила, что компания Netflix выбрала AWS для запуска различных критически важных, ориентированных на клиента и серверных приложений. Несмотря на то что Amazon Web Services заботится о технологической инфраструктуре, Netflix продолжает работать над тем, чтобы подписчикам было как можно удобней смотреть сериалы и фильмы на телевизоре и компьютере, а также получать DVD по почте».*

Далее говорится, что компания Netflix уже более года использует AWS для экспериментов по созданию рабочих процессов, а это означает, что с 2009-го Netflix находится в фазе перехода к cloud native. Очевидно, что компания Netflix увидела

перспективу и начала активно использовать преимущества нового стиля вычислений с самого начала — с тех пор как в 2006 году AWS выпустила свой первый сервис.

Со временем они осуществили поэтапную миграцию компонентов, чтобы снизить риск, получить опыт и использовать инновации, которые предлагала компания AWS. Вот краткий график их миграции: http://www.sfisaca.org/images/FC12Presentations/D1_2.pdf (2009–2010), https://www.slideshare.net/AmazonWebServices/ent209-netflix-cloud-migration-devops-and-distributed-systems-aws-reinvent-2014 (2011–2013, слайд 11) и https://medium.com/netflix-techblog/netflix-billing-migration-to-aws-451fba085a4 (2016).

- 2009 год — миграция главной системы видеоконтента в AWS S3.
- 2010 год — DRM, маршрутизация CDN, веб-регистрация, поиск, выбор перемещения, метаданные, управление устройствами и многое другое перенесены в AWS.
- 2011 год — обслуживание клиентов, международный поиск, журналы вызовов и аналитика обслуживания клиентов.
- 2012 год — страницы поиска, E-C и ваша учетная запись.
- 2013 год — большие данные и аналитика.
- 2016 год — биллинг и платежи.

Вы можете узнать больше об этом, перейдя по адресу https://media.netflix.com/en/company-blog/complete-the-netflix-cloud-migration. Это семилетнее путешествие позволило компании Netflix закрыть собственные центры обработки данных в январе 2016 года, и теперь она является cloud native-компанией. По общему признанию, этот переход для Netflix был нелегким и на пути пришлось принять множество трудных решений, что будет справедливо для любой миграции в облако. Однако долгосрочные преимущества реорганизации системы с архитектурой cloud native вместо простого переноса текущего состояния в облако означают, что все технические ограничения остались позади. Поэтому, по словам Юрия Израилевского (вице-президент по облачным и платформенным разработкам в Netflix), *«мы выбрали cloud native-подход, перестроив практически все наши технологии и коренным образом изменив методы работы компании. Архитектурно мы перешли от монолитного приложения к сотням микросервисов и преобразовали нашу модель данных с использованием СУБД NoSQL. Утверждение бюджета, централизованная координация выпуска и многонедельные циклы предоставления оборудования сменились непрерывной поставкой, команды разработчиков принимают независимые решения, используя инструменты самообслуживания в слабо связанной среде DevOps, помогая ускорять инновации»*.

Это удивительное путешествие для Netflix продолжается по сей день. У модели CNMM нет конечной точки; она эволюционирует вместе с облачными архитектурами и компаниями, которые находятся на пике развития этих архитектур.

Преимущества

В процессе перехода в облако компания Netflix проделала впечатляющий путь, и это продолжает приносить выгоду как ей самой, так и ее клиентам. Рост Netflix начался примерно в 2010 году, не замедляясь в дальнейшем, что затрудняло для нее логистическое обеспечение потребности в дополнительном оборудовании, а также запуске и масштабировании своих систем. В копании быстро осознали, что их основной приоритет — создание и распространение развлекательного контента, а не обслуживание центров обработки данных. Зная, что управление постоянно растущим числом центров обработки данных по всему миру будет продолжать вызывать огромный отток капитала и требовать сосредоточенности, которая не является приоритетной для их клиентов, они приняли решение «облако превыше всего».

Гибкость облака, возможно, — ключевое преимущество для Netflix, поскольку позволяет им при необходимости добавлять тысячи инстансов и петабайты свободного места в хранилище по мере роста их клиентской базы и увеличения нагрузки. Netflix опирается на способность облака предоставлять необходимые ресурсы, что подразумевает зависимость от облачных сервисов для обработки и анализа данных, перекодирования видео, биллинга, системы платежей и многих других услуг, на которых основан их бизнес. Помимо масштабности и гибкости, облачные технологии обеспечивают существенное повышение доступности сервисов Netflix. В компании смогли использовать облако для распределения своих рабочих процессов по зонам и географическим регионам, в которых применяются принципиально ненадежные, но избыточные компоненты для достижения желаемой доступности услуг в 99,99 %.

Наконец, хотя стоимость не была основной причиной перехода в облако, *затраты компании на начало потоковой передачи* в конечном итоге составляли только часть тех расходов, которые она несла, когда управляла собственными центрами обработки данных. Это был очень полезный побочный эффект масштабирования, которого Netflix смогла достичь, и получить выгоду было возможно только благодаря эластичности облака. В частности, это позволило *«постоянно оптимизировать смешивание типов инстансов и практически мгновенно увеличивать и сокращать объем без необходимости поддерживать буферы большой емкости. Мы также можем извлечь выгоду из экономии масштабирования, которая возможна только в большой облачной экосистеме»*. Благодаря этим преимуществам Netflix может больше не тратить ресурсы на направления, не оказывающие прямого влияния на главную миссию компании — заботу о клиентах и выполнение бизнес-требований.

CNMM

Теперь, когда мы понимаем, в чем заключается переход Netflix в облако и какую пользу они от него получили, в данном разделе с помощью CNMM оценим, как все осуществляется и какого уровня зрелости достигла компания. Поскольку чаще

всего говорили о работе, которую они проделали, чтобы перевести свою систему выставления счетов и платежей на AWS, именно эти рабочие процессы будут использоваться для оценки. Данная система состояла из пакетных заданий, API для выставления счетов и интеграции с другими сервисами в стеке составных приложений, включая существующий в то время локальный центр обработки данных. Полную информацию о миграции можно найти в блоге https://medium.com/netflix-techblog/netflix-billing-migration-to-aws-451fba085a4.

Ось облачных сервисов

Основной областью применения спектра сервисов cloud native является демонстрация количества сервисов облачных поставщиков, используемых в архитектуре. Хотя весь спектр услуг, которыми пользуется Netflix, неизвестен, компания раскрыла многочисленные сервисы AWS, которые помогают ей построить свою архитектуру. Обратившись к схеме, показывающей набор услуг сложившихся облачных сервисов, в начале этой главы (см. рис. 1.3), можно заметить, что Netflix использует большинство базовых сервисов, входящих в инфраструктуру: сетевые, вычислительные, хранилища и уровни баз данных. Кроме того, используется большинство сервисов уровня безопасности и сервисов приложений. Шел разговор и об использовании множества сервисов в таких категориях, как системы управления, аналитика, разработка и искусственный интеллект. Такое количество сервисов классифицирует Netflix как очень продвинутого пользователя нативных облачных сервисов.

Важно также отметить, что компания Netflix задействует и сервисы, которые не находятся в облаке. Там открыто говорят о том, что использование *сетей доставки контента (content delivery networks, CDN)* считается основным фактором успеха бизнеса, поэтому компания создает собственную глобальную сеть контента и управляет ею. Об этом говорится в сообщении в блоге компании по адресу https://media.netflix.com/en/company-blog/how-netflix-works-with-isps-around-the-globe-to-deliver-agreat-viewing-experience (2016 год), где четко заявлено об использовании AWS и CDNs и о том, почему было принято такое решение: *«По сути, все, что происходит перед нажатием кнопки Play, происходит в AWS, включая всю логику интерфейса приложения, поиск и выбор контента, алгоритмы рекомендаций, транскодирование и т. д. Мы используем AWS для этих приложений, поскольку потребность в данном типе вычислительных систем не уникальна для Netflix, и мы можем воспользоваться простотой применения и растущей коммерциализацией рынка облачных вычислений. Все, что происходит после того, как вы нажали Play, является уникальным для Netflix, и наша растущая потребность в масштабировании в этой области позволила сделать более эффективными как процесс доставки нашего контента, так и Интернет в целом»*.

Кроме того, в некоторых случаях Netflix выбирает инструменты с открытым исходным кодом, работающие в облаке, такие как Cassandra — для своей базы данных NoSQL или Kafka — для потоков событий. Использование этих архитектурных

решений стало компромиссом, который гарантирует, что компания задействует для удовлетворения своих потребностей лучшие инструменты, а не только то, что предлагает поставщик облачных решений.

Ось проектирования, ориентированного на приложения

Разработка приложения для облака — это, пожалуй, самая сложная часть пути, и достижение высокого уровня зрелости на оси разработки, ориентированной на приложения, потребует особых подходов. Компания Netflix столкнулась с рядом серьезных проблем во время миграции в облако биллинга и платежной системы, в частности, им требовались практически нулевое время простоя, масштабируемость, соответствие SOX и глобальное развертывание. На момент начала работы над проектом у нее уже было много других систем, работающих в облаке в качестве разделенных сервисов. Поэтому она использовала тот же подход, создавая микросервисы для систем выставления счетов и оплаты.

Цитата из блога компании на эту тему: *«Мы начали разбивать существующий код на более мелкие эффективные модули и сначала переместили некоторые критические зависимости, чтобы они запускались из облака. Сначала мы перенесли в облако свое решение для налогового учета. Затем удалили историю обслуживания участников из гигантских таблиц, которые были частью множества различных экземпляров кода. Мы создали новое приложение для сбора событий выставления счетов, перенесли только необходимые данные в новое хранилище данных Cassandra и начали глобально обслуживать историю выставления счетов из облака. Мы потратили много времени на создание инструмента миграции данных, который преобразует атрибуты выставления счетов пользователям, распределенные по многим таблицам в Oracle, в гораздо более простую структуру данных Cassandra. Мы работали совместно с инженерами из DVD-подразделения, чтобы еще больше упростить интеграцию и избавиться от устаревшего кода».*

Другим важным изменением в результате этого процесса стал переход от реляционной архитектуры базы данных Oracle к более гибкой и масштабируемой структуре данных NoSQL для обработки подписки и региональной распределенной реляционной СУБД MySQL для обработки транзакций пользователя. Эти изменения требовали переработки дизайна других сервисов Netflix, чтобы можно было воспользоваться преимуществами разделения хранилища данных и дублировать ввод в их собственное хранилище на основе NoSQL. Это позволило Netflix перенести миллионы строк из локальной базы данных Oracle в Cassandra в AWS, не повлияв на пользователя.

В процессе миграции биллинга и платежной системы в облако компания Netflix приняла множество важных решений, которые повлияли на ее архитектуру. При этом учитывалось их долгосрочное воздействие, что увеличило время миграции, но подготовило архитектуру к будущим изменениям и дало ей возможность

масштабироваться по мере роста на международном уровне. Очистка кода для уменьшения технического долга — яркий пример этого, она позволила убедиться, что новая кодовая база была разработана с использованием микросервисов и соответствует другим принципам облачно-ориентированного проектирования. Компания Netflix продемонстрировала высокий уровень зрелости оси разработки, ориентированной на приложения.

Ось автоматизации

Ось автоматизации демонстрирует способность компании управлять, работать, оптимизировать, обеспечивать безопасность и прогнозировать поведение ее систем, чтобы обеспечить положительный опыт работы с клиентами. В начале перехода в облако в компании Netflix поняли, что нужно разработать новые способы проверки того, что системы работают на высочайшем уровне производительности и, что более важно, устойчивы к ошибкам всех видов. Они создали набор инструментов под названием Simian Army (https://medium.com/netflixtechblog/the-netflix-simian-army-16e57fbab116), включающий в себя все виды автоматизации, которая используется для выявления узких мест, контрольных точек и многих других проблем, способных помешать клиентам. Одним из оригинальных инструментов и источником вдохновения при разработке всего набора Simian Army является их приложение Chaos Monkey, о котором говорят следующее: «*...наша философия, которой мы придерживались, создавая Chaos Monkey — инструмент, который случайным образом отключает наши промышленные инстансы, чтобы гарантировать, что мы сможем пережить подобный сбой, никак не повлияв на клиента. Название происходит от идеи выпустить дикую обезьяну с оружием в вашем ЦОД (или облачном регионе), чтобы она хаотично стреляла и грызла кабели, — все это время мы без перерывов продолжаем обслуживать своих клиентов. Запустив Chaos Monkey в середине рабочего дня в тщательно контролируемой среде в присутствии инженеров, готовых решить любые проблемы, мы можем выявить слабые стороны нашей системы и создать механизмы автоматического восстановления для их устранения. Так что, если в следующий раз сбой произойдет в три часа ночи в воскресенье, мы его даже не заметим*».

Наличие систем, способных продолжать работу при случайном отключении критически важных сервисов, определяет высокий уровень автоматизации. Это означает, что все системы должны соответствовать строгим автоматизированным процессам, включая управление средой и конфигурацией, развертывание, мониторинг, соблюдение нормативных требований, оптимизацию и даже прогнозирование. Приложение Chaos Monkey вдохновило на создание многих других инструментов из набора Simian Army, который включает:

- *Latency Monkey* — специально задерживает обработку REST-вызовов, чтобы имитировать деградацию сервиса (ухудшение качества обслуживания);
- *Conformity Monkey* — находит инстансы, которые не соответствуют заранее определенным требованиям, и закрывает их;

- *Doctor Monkey* — подключается к проверкам, которые выполняются в инстансе для мониторинга внешних признаков работоспособности;
- *Janitor Monkey* — ищет неиспользованные ресурсы и избавляется от них;
- *Security Monkey* — обнаруживает нарушения безопасности и уязвимости, а также устраняет скомпрометированные инстансы;
- *10–18 Monkey* — обнаруживает проблемы конфигурации и времени выполнения в определенных географических регионах;
- *Chaos Gorilla* — похож на Chaos Monkey, но имитирует полное отключение доступных зон AWS.

Однако на этом специалисты не остановились. Они создали также облачную платформу для телеметрии и мониторинга Atlas (https://medium.com/netflix-techblog/introducing-atlasnetflixs-primary-telemetry-platform-bd31f4d8ed9a), которая отвечает за непрерывный сбор данных. Основная цель Atlas — поддерживать запросы к данным многомерных временных рядов, чтобы они могли как можно быстрее детализировать проблемы. Этот инструмент соответствует методологии проектирования под названием «12-факторное приложение» в отношении ведения журнала и позволяет компании хранить огромные объемы данных и событий для анализа и принятия упреждающих мер. В дополнение к платформе Atlas в 2015 году компания Netflix выпустила инструмент Spinnaker (https://www.spinnaker.io/), который представляет собой открытую многопользовательскую платформу с непрерывной доставкой для быстрого и достоверного выпуска изменений программного обеспечения. Компания Netflix постоянно обновляет инструменты автоматизации и выпускает дополнительные, которые помогают ей развертывать и контролировать все свои сервисы, а также управлять ими, используя глобально распределенные области AWS, а в некоторых случаях и сервисы других облачных поставщиков.

Компания Netflix автоматизировала все процессы в своей среде, пока переносила в облако рабочие нагрузки. Сегодня компания полагается на эти инструменты, обеспечивая функционирование глобальной сети и должным образом обслуживая своих клиентов. Таким образом, она достигла очень высокого уровня оси автоматизации.

Резюме

В этой главе мы выяснили, что является средой cloud native и какие области необходимы для разработки развитой облачной архитектуры. Используя CNMM, мы определили, что все архитектуры будут базироваться на трех принципах проектирования: применении облачных сервисов, автоматизации в той или иной степени и проектировании, ориентированном на приложения. Эти принципы используются для оценки зрелости компонентов архитектуры в том смысле, в котором она к ним

применима, и в соответствии с их собственной шкалой. Мы рассказали о том, что представляет собой переход в облако для компании, как она принимает решение использовать подход «облако превыше всего», как меняет своих сотрудников, процессы и технологии, как создает облачную операционную среду и, наконец, как переносит или реорганизует свои рабочие процессы, чтобы закрепиться в облачно-ориентированном мире.

Следующую главу начнем с глубокого погружения в инфраструктуру применения облачных вычислений и постараемся более детально рассмотреть переход в облако, предпринятый компанией, изучив семь компонентов этого процесса. Мы исследуем миграции и разработку пути с чистого листа, а в завершение поговорим о безопасности и рисках, сопутствующих внедрению облака.

2 Процесс перехода в облако

Превращение в компанию cloud native — это целое путешествие, сосредоточенное не только на технологиях. Как показывает пример Netflix, переход может занять длительное время и потребовать принятия трудных решений, связанных в том числе с техническими и коммерческими компромиссами. Кроме того, это путешествие бесконечно. Облака все еще находятся в зачаточном состоянии, и инновации от крупных поставщиков облачных услуг только набирают обороты. В данной главе определены основные стимулы для перехода в облако. В ней также рассмотрены структуры, которые организации часто используют в процессе миграции, и объясняется, какие компоненты и подходы при этом применяются. В конце будет показано, как создать комплексную облачную операционную модель с учетом рисков, проблем с безопасностью и обеспечения качества в условиях облака.

Стимулы для перехода в облако

Переход к работе в облаке происходит не случайно и требует принятия осознанного решения. Однако это решение будет лишь отправной точкой в длительном процессе, для которого нужно много человеческих ресурсов и технологических изменений. Причины, по которым организации склоняются к такому переходу, могут быть разными, но чаще всего основными факторами являются скорость и экономия денежных средств. Есть и другие важные аспекты, которые принимают во внимание, включая безопасность и управление активами компании, региональное или международное расширение деятельности, а также желание привлечь самых талантливых работников или воспользоваться последними технологическими инновациями. Эти стимулы определяют процесс перехода в облако для компаний любых размеров, и в этом разделе вы узнаете, почему они так важны и каким образом влияют на принятие решений.

Быстрое продвижение и низкие траты

Еще до появления облачных вычислений при проектировании системы необходимо было оценивать требования к производительности и затем выделять соответству-

ющие аппаратные ресурсы. Это дорогостоящий и медленный процесс, который часто протекал в условиях нехватки информации. Он приводил как к принятию неудачных ценовых решений, так и к появлению лишних ресурсов, которые простаивали без дела в вычислительных центрах. Более того, если предполагалось создать совершенно новое бизнес-направление, само выполнение подобных расчетов могло оказаться невозможным. Таким образом, для компании, пытающейся заниматься инновациями в облаке, успех могут обеспечить минимальные начальные инвестиции и нацеленность на экспоненциальный рост. Покупка или аренда оборудования у поставщика, с которым вы давно сотрудничаете, не имеет ничего общего с облачными вычислениями. Облако умеет выделять ресурсы на лету и удалять их, когда они больше не нужны.

Гибкость

Достижение гибкости часто называют основным стимулом для перехода в облако. Распространенная проблема, с которой компании сталкиваются на протяжении десятилетий, — длительное время развертывания оборудования в их центрах обработки данных, что создает всевозможные проблемы, включая затягивание работы над проектом. Устранение этого ограничения за счет наличия практически безграничной емкости и возможности задействовать ее в течение всего нескольких минут имеет решающее значение для организаций с высокими требованиями к скорости ведения бизнеса. Интернет и социальные сети ускоряют обмен идеями, и компании будут охотиться за этими идеями и пытаться монетизировать их в своих бизнес-моделях. Поскольку часто уже через несколько месяцев идеи теряют свою привлекательность, длительное ожидание физических ресурсов означает упущенную выгоду от их реализации. В сфере облачных вычислений этой проблемы не существует, организации могут разрабатывать рыночные стратегии, исходя из своей основной специализации, позволив облачному поставщику позаботиться о материальных ИТ-ресурсах.

Fail-fast (завершение работы при первой ошибке) — это еще одна область, где важную роль играют экспериментирование и скорость/гибкость облака. В наши дни дискредитация идей и распад компаний происходят в рекордные сроки, и, чтобы избежать неудачи, компания должна постоянно совершенствовать свою бизнес-модель, максимально приближая ее к требованиям клиентов. Вот слова Джеффа Безоса о концепции эксперимента: *«Если вы знаете, что эксперимент сработает, это уже не эксперимент»*. Это означает, что очень важно постоянно и быстро пробовать новые идеи и создавать продукты и услуги. Еще это означает, что многие идеи не достигнут желаемых бизнес-целей, и это нормально, так как только некоторые из них окажутся успешными. Облачные вычисления обеспечивают гибкость за счет возможности создавать и удалять ресурсы в нужный вам момент. Таким образом, если как следует спроектировать систему минимального размера, она сможет расти экспоненциально по мере роста нагрузки, благодаря чему у компании никогда не будет простаивающего или редко используемого оборудования.

Денежные затраты

Кроме всего прочего, fail-fast позволяет сократить денежные расходы. Если эксперименты стоят дорого, их неудачный итог может иметь негативные последствия для компании. Как только проект или система начинают существенно влиять на бизнес, сразу же реализуется система сдержек и противовесов, которая замедляет их рост. Это позволяет предотвратить принятие затратного решения, такого как закупка дорогого аппаратного обеспечения при низкой вероятности окупаемости. Идея гибкой разработки вовсе не нова; она существует на протяжении десятилетий и не имеет прямого отношения к облаку. Однако облачные вычисления делают ее выгодной с точки зрения расходов, требуя минимальных первоначальных вложений. Полный жизненный цикл новой бизнес-идеи может занимать недели или месяцы, и на первых порах важно ограничить использование облачных ресурсов. Если идея станет популярной, архитектура cloud native позволит системе масштабироваться с ростом нагрузки. Если желаемый результат не будет достигнут, можно легко остановить ее и списать те небольшие ресурсы, которые в нее вложили, на невозвратные потери. Вот что означает fail-fast при разработке инноваций в облаке.

Финансовая привлекательность не ограничивается использованием минимальных ресурсов для развертывания новых систем. Большинство облачных поставщиков по-новому подходят к тарификации ресурсов. Самые крупные из них обычно имеют очень гибкие тарифы, которые иногда предусматривают посекундную (или еще более тонкую) оплату с выставлением счета раз в месяц. Ситуации, когда поставщик требовал, чтобы компании покупали ресурсы, а затем взимал существенную плату за их обслуживание, в прошлом. Сейчас в цену облачных ресурсов все включено, к тому же они автоматически улучшаются, когда поставщик выполняет обновление. Ежемесячная оплата — огромный плюс для большинства организаций, которые могут относиться к ней как к операционным расходам, когда не требуется одобрение долгосрочных капитальных затрат. Этот вопрос относится в основном к финансовому учету, но является и важным аспектом гибкости, которую компания приобретает благодаря переходу в облако, так как при этом новые идеи и системы не должны ждать одобрения или включения в список капитальных расходов.

Обеспечение безопасности и надлежащей управляемости

На протяжении длительного времени организациям разных размеров приходилось использовать собственные вычислительные центры, чтобы иметь возможность контролировать безопасность и управлять данными и рабочими процессами. Но благодаря облачным вычислениям наконец появилось решение, обеспечивающее более высокие безопасность и управляемость по сравнению с теми, которых мы можем достичь самостоятельно. Дело в том, что поставщики облачных услуг считают своей обязанностью предоставлять безопасные сервисы, соответствующие всем

стандартам и нормам, чтобы их клиенты могли сосредоточиться на решении своих бизнес-проблем. Поскольку поставщики облачных услуг постоянно улучшают свои сервисы, при возникновении нового шаблона безопасности, требований со стороны правительства или других важных аспектов они работают над тем, чтобы быстро адаптировать к ним свое облако. Таким образом, будучи их клиентом и используя их технологии, компания автоматически получает все преимущества от того, насколько масштабно и целенаправленно эти поставщики занимаются разработкой систем безопасности и управления, и при этом обычно не несет дополнительных затрат.

Безопасность

Каждая организация уникальна и имеет собственные требования к безопасности и управляемости, а также подходы к администрированию и защите своих активов. Облачные поставщики строят свой бизнес и формируют репутацию, гарантируя, что предоставляют услуги наиболее безопасным способом. И хотя некоторые компании по-прежнему считают, что лучше выполнять свои рабочие процессы в собственных вычислительных центрах, использование закрытой облачной архитектуры при надлежащей конфигурации облака, вне всяких сомнений, более безопасно. Акцент на безопасности только усиливается по мере того, как поставщики становятся все более совершенными и реализуют все больше инноваций в сфере защиты своих услуг. У крупных поставщиков облачных услуг есть сотни или даже тысячи сотрудников, ежедневные обязанности которых заключаются в повышении безопасности управления их окружениями, разработке новых способов масштабирования механизмов шифрования, защите работы с данными и обеспечении более безопасного выполнения рабочих задач на их платформах тем или иным способом. Благодаря совершенствованию методик искусственного интеллекта и машинного обучения облачная безопасность экспоненциально растет, это выражается не только в упреждающем мониторинге, но и в автоматическом обнаружении потенциальных уязвимостей, исправлении неправильно сконфигурированных компонентов и всевозможных мерах защиты данных и систем.

Наверное, все еще можно найти компании или организации, которые похвастаются похожим уровнем целенаправленности в этой важной сфере, однако абсолютное большинство не может позволить себе выделить ресурсы, необходимые для постоянного улучшения безопасности, как это делают поставщики облачных услуг. Более того, этот же аргумент в пользу применения облаков справедлив и когда нужно ответить на следующий вопрос: даже если вы способны сравниться с поставщиками облачных услуг по количеству внедренных инноваций, оптимально ли тратить на это ресурсы организации или стоит сосредоточиться на бизнес-результатах? Довод в пользу обеспечения безопасности с помощью облака почти настолько же силен, насколько сильны преимущества в плане денежных расходов и адаптируемости. Каждой организации необходимо очень подробно обсудить требования к безопасности и управляемости, прежде чем принимать решение, куда инвестировать — в локально размещаемое оборудование или закрытое облако. Часто для этого

нужно пристально следить за командами, которые участвуют в обсуждении, чтобы они не пытались уцепиться за знакомые технологии, используя безопасность или управляемость в качестве предлога и создавая тем самым риски для окружающих.

Управляемость

Одно дело — исходить из того, что облако безопасно, и совсем другое — уметь воспользоваться этим преимуществом, чтобы обеспечить уверенное положение своей компании. Поставщики облачных услуг предлагают множество сервисов, отвечающих практически любым требованиям к безопасности. Однако реализация этих сервисов в соответствии с проектной спецификацией ложится на конкретную систему. И после этого еще необходимо убедиться в том, что реализованные процедуры соблюдаются и поддерживаются, а это требует хорошей управляемости. Облачные сервисы взаимодействуют между собой, формируя слой мониторинга, аудита, управления конфигурацией и сопутствующих активностей. Использование облака позволит задействовать в рабочих заданиях, развертываемых компанией, одну и ту же модель управления, не игнорируя этот аспект из-за его сложности или больших расходов, что иногда случается при локальном развертывании.

Расширение компании

Для многих компаний рост является первоочередным приоритетом наряду с обслуживанием существующих клиентов, увеличением доли на рынке и исследованием новых бизнес-направлений. Расширение может быстро превратиться в существенную статью расходов, если возникнет необходимость в развертывании новых вычислительных центров. Организация вычислительного центра возможна, только если у компании есть хорошее технико-экономическое обоснование его прибыльности, но даже в этом случае требуются существенные капитальные затраты. Потраченное время, денежные вложения и риски, связанные с развертыванием нового вычислительного центра, оказываются значительными, и все больше компаний начинают сомневаться в том, насколько этот вариант жизнеспособен. Риски повышаются еще больше, когда расширение происходит за пределами традиционных рынков: в новом городе в том же географическом регионе или в другой стране.

Использование облака для такого расширения — идеальный способ достичь желаемых результатов при значительно меньших начальных капиталовложениях. И неважно, требуется это расширение на другом конце страны или мира, — большинство крупных поставщиков облачных услуг работают в местах, где можно обслуживать множество потребителей. Если компания заранее уделит достаточно внимания автоматизации рабочих задач, их развертывание в дополнительных облачных регионах будет довольно простым. Выход на глобальный рынок в считаные минуты — это большое рыночное преимущество крупных поставщиков облачных услуг, и любая компания, развертывающая свои рабочие задания, должна это учитывать.

Привлечение и удержание талантливых сотрудников

Компаниям, переходящим в облако, нужны талантливые люди, которые разбираются в технологиях и новых процессах, сопутствующих переходу. Во многих случаях компании ориентируются на удержание своих сотрудников и их переквалификацию. Этот процесс имеет множество преимуществ, основное из которых — то, что все профессиональные знания этих людей остаются внутри организации. С переходом в облако бизнес-процессы и подходы часто становятся похожими, поэтому наличие информации о том, почему были приняты те или иные решения в ходе изначальной реализации, имеет огромное значение, — это позволяет убедиться в том, что система спроектирована правильно.

Привлечение новых талантов, как и удержание старых, будет играть важную роль в долгосрочном успехе перехода в облако. Нанимая специалистов в облачных вычислениях, вы получаете существенные преимущества, которые помогут привить сотрудникам компании новый образ мышления, так как теперь они не будут стеснены ограничениями, характерными для вычислительных центров. Это касается не только технических специалистов. Бизнес-профессионалы, осознающие все трудности, связанные с миграцией, помогут компании пройти этот процесс. Глобальная битва за самых талантливых работников уже в разгаре, и по мере того, как все больше и больше компаний станут выбирать этот путь, спрос будет только расти. Это критически важный аспект, который следует учитывать любой компании, ориентированной на переход в облако.

Облачные инновации и экономия на масштабе

Чтобы успешно перейти в облако, лучше всего направить ресурсы на свои основные конкурентные преимущества и пользоваться инновациями, которые внедряются крупными поставщиками облачных услуг с максимально возможной скоростью. Очень немногие компании могут считать управление вычислительными центрами своей сильной стороной, это должно быть основным принципом во время принятия решения о внедрении технологий cloud native. Этот всеохватывающий принцип только стимулирует переход в облако, поэтому его следует считать большим преимуществом для любой организации. Облачные инновации помогут компаниям лучше адаптироваться, снизить расходы, обезопасить рабочие процессы, выйти на новые рынки и привлечь/удержать лучших сотрудников, соответствующих потребностям организации.

Помимо высоких темпов появления инноваций, эти облачные поставщики работают в огромных масштабах, что позволяет им вести переговоры о ценах со своими партнерами на условиях, недоступных любым другим компаниям, какими бы большими те ни были. Это справедливо для затрат на вычисления, электроэнергию, накопители, сети, площади для новых вычислительных центров и даже наем талантливых работников, которые помогают создавать эти инновации. Организации,

рассматривающие возможность перехода в облако, должны как следует подумать об экономии на масштабе: если они не могут существенно повлиять на ценообразование, цены, предлагаемые облачными поставщиками, всегда будут выгодней.

Операционная облачная модель

После того как организация оценит стимулы к переходу в облако и примет соответствующее решение, начнется настоящая работа. В чем она заключается? Согласно Cloud Adoption Framework от Amazon Web Services (d0.awsstatic.com/whitepapers/aws_cloud_adoption_framework.pdf), *«переход в облако требует обсуждения и рассмотрения фундаментальных изменений в рамках всей организации и поддержки заинтересованных лиц в каждом подразделении — как внутри, так и за пределами ИТ»*. Более того, в ходе миграции по-прежнему следует фокусироваться на трех областях: людях, процессах и технологиях, но это слишком большое упрощение, учитывая масштаб происходящих изменений. Этот процесс предусматривает участие владельцев компании и специалистов по набору персонала, изменения в закупках, разработку требований к управлению проектами и строгий контроль над ним. Кроме того, новые технологии станут по-разному влиять на все вовлеченные стороны, а конкретные решения, касающиеся целевой платформы, безопасности и системного администрирования, будут критически важными.

Существует множество теорий организационных изменений, которые можно приурочить к переходу в облако, и, как любая трансформация, это потребует участия всех заинтересованных лиц из всей организации. Одним из подходов к организационным изменениям, который хорошо сочетается с миграцией в облако, является теория Джона Коттера (www.kotterinternational.com/8-steps-process-for-leading-change/). Коттер определил восемь шагов, обеспечивающих успешную реализацию нисходящих изменений на предприятиях.

1. Создайте ощущение безотлагательности.
2. Сформируйте команду реформаторов.
3. Выработайте стратегическое видение и инициативы.
4. Наберите армию добровольцев.
5. Устраните препятствия, чтобы двигаться вперед.
6. Добивайтесь краткосрочных побед.
7. Закрепляйте достигнутые успехи и углубляйте перемены.
8. Укореняйте изменения в корпоративной культуре.

Эта методология хорошо подходит для перехода в облако, так как изменения обычно оказываются настолько значительными, что их необходимо начинать сверху, часто с совета директоров и генерального директора. На этом уровне цели, выражающиеся

в простой экономии или адаптируемости, оказываются слишком приземленными, желаемым результатом должно стать изменение в бизнесе, которое задаст для компании новый курс к увеличению доходов и стоимости акций. Таким образом, привязка этого перехода к бизнес-изменениям создает ощущение неотложности, без которой часто нельзя обойтись. В итоге формируется коалиция сторонников такого видения, которые будут его продвигать. В числе первых добровольцев будут те, кто имеет непосредственное отношение к работе. Это люди с высокой производительностью труда, готовые идти на риск; они виспринимают эти изменения как возможность поближе познакомиться с новой технологией, продвинуться вверх по карьерной лестнице или стать частью чего-то большого. Устранение барьеров позволит им быстро продвигаться вперед и оставаться сосредоточенными. Чтобы добиться долгосрочных успехов, придется заинтересовать менее авантюрных людей; для этого потребуется целый ряд небольших достижений, которые докажут состоятельность выбранной стратегии, что в конечном счете приведет к ускорению развития новых или переноса уже существующих облачных проектов, которые в итоге станут новой нормой.

Заинтересованные лица

Переход в облако состоит не только в реализации новых технологий, оно подразумевает также адаптируемость к бизнес-изменениям и все прочие стимулы, о которых шла речь в начале главы. Поэтому список заинтересованных лиц, на которых этот переход повлияет и которые по этой причине должны быть вовлечены во все его аспекты, может быть довольно длинным. Почти всегда тем или иным образом вовлеченным оказывается все высшее руководство, так как изменения трансформируют всю компанию. Кроме того, важную роль будут играть руководители подразделений и сами подразделения, так как приложения обычно предназначены для удовлетворения их нужд, связанных с продажами или доставкой. Наконец, нельзя обойтись без ключевого участника — ИТ-отдела: он будет интегрироваться во все процессы, протекающие в компании, продвигая облачные технологии и связанные с ними изменения.

Во многих организациях в ИТ-отделе есть несколько крупных руководителей, в частности директор по информационным технологиям или вице-президент по корпоративным системам, технический директор или вице-президент по бизнес-приложениям, вице-президент по вычислениям для конечных пользователей или поддержке, вице-президент по инфраструктуре и директор по ИТ-безопасности, в круг обязанностей которых входят все существующие функции. Переход в облако не устраняет эти роли, но в некоторых случаях меняет их назначение. Например, вице-президент по инфраструктуре часто становится вице-президентом по DevOps или кем-то наподобие, так как требования к управлению инфраструктурой переходят из физической формы в код. Одно это изменение может высвободить значительное число сотрудников, занятых обслуживанием вычислительных центров или закупкой оборудования, и сориентировать их на разработку бизнес-приложений. Помимо этой большой трансформации, всем остальным членам ИТ-команд придется привести

свои навыки в соответствие с выбранной облачной платформой, что часто требует активного обучения и правильного посыла со стороны руководства.

В этом процессе бизнес-аспекты зачастую считаются ключевыми, но что на самом деле это означает? В этом случае коммерческая часть организации приносит доход и владеет продуктами или услугами от имени компании. В больших международных корпорациях бизнес может быть организован в виде отдельных дочерних компаний с множеством подразделений, работающих в каком-то определенном направлении. В организациях поменьше работники коммерческих отделов приносят доход, постоянно предлагают новые идеи и пытаются вывести их на рынок раньше своих конкурентов, что, в сущности, и есть *гибкость бизнеса*. Любая компания должна стараться выделять как можно больше ресурсов на то, чтобы сделать бизнес более гибким и, как следствие, получить конкурентное преимущество. Поэтому, если ИТ-отделу удастся переориентировать людей с рутинных технических задач на проектирование или реализацию бизнес-приложений, это напрямую отразится на доходах компании.

Управление изменениями и проектами

Когда организация берется за переход в облако, она должна подумать о том, какие процессы будут затронуты. Особенно это касается управления изменениями и проектами. Эти два важных аспекта операционной модели ИТ должны учитываться при переходе. При этом обычно приходится менять политику их обеспечения в облаке. Если тщательно исследовать управление изменениями во многих организациях, можно заметить, что, несмотря на обширность и продвинутость, многие процессы замедляют развертывание ИТ-ресурсов, снижая темпы ведения бизнеса ради минимизации рисков. Эти процессы создавались вовсе не с целью замедлить работу, но со временем по мере обнаружения нестандартных ситуаций или упущений они бюрократизировались. Постепенно добавлялись новые требования, более продолжительные планы отката изменений и другие инициативы, ухудшающие гибкость бизнеса. Одним из побочных эффектов замедления является то, что, прежде чем сделать существенное капиталовложение, выполняется глубокий анализ, чтобы обеспечить соответствие процессов требованиям и бюджету проекта.

Облако не освобождает от необходимости в строгих процессах управления изменениями. Однако его наличие меняет подходы к их реализации таким образом, который может повысить их эффективность и при этом избавиться от бюрократии, замедляющей работу. Архитектуры cloud native по своей природе имеют меньшую связанность и больше ориентированы на сервисы, что устраняет необходимость в развертывании крупных проектов одним махом и ускоряет доставку кода. Переход в облако может не только избавить вас от технического долга в виде устаревшего кода, но и убрать устаревшие процессы. Например, ITIL, распространенный стиль организации работы и управления изменениями, используется на многих предприятиях и предназначен для приближения ИТ-услуг к нуждам бизнеса. ITIL предусматривает строгие процессы реализации изменений, которые включают в себя

создание документации, цепочки принятия решений, планы отката изменений и прочие процедуры. Все это будет актуально и в облаке. Однако темпы изменений повысятся, так как объемы развертываний часто уменьшаются, а риски, связанные с изменениями, существенно снижаются. Обычно это становится результатом появления новых идей в управлении изменениями и проектами организации.

Управление изменениями

Наличие собственных шаблонов проектирования облачных приложений, особенно состоящих из множества небольших сервисов, — не единственная причина, по которой управление изменениями происходит быстрее; автоматизация и контейнеризация также являются ключевыми факторами. Это явление не уникально для облака. Но, поскольку облако по своей природе ориентировано на использование API и имеет практически неограниченные возможности сбора, хранения и анализа данных, оно заставляет нас полностью переосмыслить процесс управления изменениями. Автоматическое развертывание с использованием принципов DevOps а также конвейеры *непрерывной интеграции, непрерывного развертывания (continuous integration, continuous deployment, CICD)* делают процесс развертывания и отката кода бесперебойным и, что более важно, согласованным. Облачные поставщики владеют инструментами, которые изначально решают многие проблемы, включая организацию хранения, сборки, развертывания и тестирования кода. Кроме того, существуют способы создания усовершенствованных конвейеров CICD с применением пользовательских методов разработки.

Управление проектами

С развитием методов управления изменениями при переходе в облако должен совершенствоваться и подход к управлению проектами. Каскадная методология, при которой все требования определяются заранее, а затем разрабатываются последовательно с завершением циклов тестирования, слишком медленна для обеспечения нужной гибкости бизнеса. В прошлом, когда выделение аппаратных ресурсов или отсутствие командного взаимодействия снижали производительность труда, каскадный метод прекрасно работал, что позволяло создавать высококачественные системы. С устранением этих барьеров в облаке стала чаще использоваться гибкая методология. Этот стиль позволяет параллельно выполнять многие действия, включая сбор требований, разработку, тестирование и развертывание. Возможность облака мгновенно выделять ресурсы и увеличивать их объем, когда это необходимо, делает такой стиль управления проектами действительно хорошим. Скорость, с которой бизнесу необходимо изменять требования в связи с изменением рыночных условий, может быть высокой, а гибкое управление проектами позволяет быстро изменять требования без значительных переделок.

При изменении процессов управления проектами с учетом принципов проектирования облачных вычислений одним из важных факторов будет включение передовых

методов разработки облачных технологий. Добавление в процессы проектирования проверок обеспечения качества, которые выявляют лучшие практики разработки масштабируемых безопасных архитектур cloud native, гарантирует, что рабочие нагрузки будут развернуты готовыми к масштабированию в облаке и потребуют меньшего количества исправлений после запуска в эксплуатацию. Включив практики облачного проектирования в процессы управления проектами, компания раньше начнет разработку архитектур cloud native, сосредоточится на автоматизации и инновациях от поставщиков облачных услуг и сможет быстро менять направление в зависимости от потребностей бизнеса. Эти изменения не только увеличивают скорость бизнес-процессов, но и снижают риски.

Риск, соответствие требованиям и обеспечение качества

При переходе компании в облако повышается ее профиль риска, по крайней мере на начальном этапе. Это происходит из-за появления новых источников проблем, а не потому, что облако — рискованное место. После того как активы компании (например, данные и код) перенесены на вычислительные ресурсы, находящиеся вне сферы ее прямого контроля, должны появиться дополнительные соображения относительно того, как обеспечить безопасность этих ресурсов. Соблюдение отраслевых стандартов соответствия — важный первый шаг к соблюдению основных требований к данным, собираемым компанией. Кроме того, управление процессами и их аудит гарантируют, что не ускользнет ничего, что может вызвать утечку данных или несанкционированный доступ к активам компании. Наконец, тестирование обеспечения качества не только системного кода и функциональных возможностей, но и инфраструктуры, средств управления безопасностью и других процедур и планов для обеспечения высокой доступности и непрерывности бизнес-процессов — обязательное условие для компаний cloud native.

Все эти технические аспекты зачастую сильно различаются при сравнении локально размещенных и облачных систем, и для компаний крайне важно корректировать свою работу в облаке, а также привести свои локальные процессы в соответствие с облачными. Основная цель перехода в облако, как и любого другого, — изменить способ использования ИТ-ресурсов и устранить локальные ограничения при разработке приложений. Это означает, что даже для выполняемых локально операций эти идеи должны реализовываться так, чтобы при переносе устаревших рабочих процессов в облако они уже соответствовали обновленным принципам проектирования. Со временем, когда в результате миграции и новых разработок компания начнет в первую очередь ориентироваться на облако, подход к рискам, соответствию требованиям и обеспечению качества также изменится. Технологические соображения на самом деле — одна из самых простых составляющих операционной модели. Несмотря на то что это сложный и часто новый процесс, он носит предсказуемый характер и предоставляет множество возможностей для приобретения новых знаний и навыков.

Риск и соответствие требованиям

Когда компания работает с данными, обрабатывает транзакции или взаимодействует со своими клиентами, риски неизбежны. Возникают рискованные ситуации, поскольку всегда найдутся недобросовестные люди, которые захотят похитить данные или повлиять на транзакции или взаимодействие с пользователями. Но злоумышленники — не единственная проблема, с которой сталкивается компания. Сбои в вычислениях или программные ошибки более чем реальны, и любая компания должна уметь справляться с ними. Облако предлагает изменение уровня риска, но не его снижение. Большинство крупных облачных провайдеров работают по модели совместной ответственности за реализацию мер безопасности и контроля. Как правило, это означает, что облачный провайдер владеет (и принимает на себя риск) чем-то конкретным для супервизора и управляет им, в то время как клиенты отвечают за всю операционную систему вплоть до стека приложений. Обычно облачные провайдеры крайне настойчивы в получении различных отраслевых и государственных сертификатов соответствия (например, Стандарта безопасности данных индустрии платежных карт или PCI-DSS, ISO 27001) для своих сервисов. Риск, связанный с этим, заключается в следующем: компания должна понимать, что в соответствии с моделью совместной ответственности услуги облачных провайдеров могут быть сертифицированы для определенного типа рабочей нагрузки, но если рабочая нагрузка спроектирована компанией без учета реализации этих элементов управления, она не сможет пройти аудит.

Соответствие требованиям — это не только внешняя проблема, часто в компании предусмотрена строгая модель внутреннего управления, которая требует наличия особых механизмов контроля определенных типов данных и рабочих нагрузок. В таких случаях поставщик облачных услуг не сможет реализовать данные требования, поскольку они специфичны для организации. Однако существуют способы сопоставить внутренние средства контроля с имеющимися сертификатами с аналогичными требованиями, чтобы продемонстрировать соответствие. Сопоставление внешних и внутренних элементов управления имеет решающее значение для успешной работы в облаке, так как гарантирует, что данные и рабочие нагрузки используют сервисы, соответствующие классификации конкретного приложения. Крупные организации часто имеют сотни или тысячи приложений, и каждое из них будет характеризоваться определенными типами данных и методами обработки, поэтому выполнение соглашения о соответствии — важное условие облачной операционной модели.

Важным соображением, которое мы обсудим в дальнейшем, является то, что темпы инноваций, внедряемых поставщиками облачных услуг, скорее всего, будут опережать способность компании анализировать и утверждать новые сервисы для использования в новых рабочих нагрузках. Хотя крайне важно, чтобы компания тщательно следила за тем, чтобы сервисы, используемые в их системах, были безопасными и могли проходить внешний и внутренний аудит, также важно, чтобы эти проверки не замедляли реализацию собственных бизнес-требований.

Инновации, предлагаемые новыми сервисами, можно использовать безопасно и своевременно, и компании должны к этому стремиться. Одним из примеров может послужить предоставление команде, занимающейся инновациями, возможности сосредоточиться на новых разработках использующих самые современные сервисы cloud native, для не слишком критических систем, в то же время продолжая разработку критически важных систем на более продвинутых и обкатанных сервисах.

Обеспечение качества и аудит

Когда компания начнет работать над соблюдением нормативных требований и снижением рисков, появятся определенные методы защиты, которые позволят ей развертывать облако. Но эти методы защиты и управления эффективны только тогда, когда они разработаны и внедрены. Постепенно, по мере того как рабочие нагрузки будут создаваться и развертываться в облаке, компания будет вынуждена повторно оценивать уровень риска и обеспечивать его соответствие текущим условиям. Со временем поставщики облачных услуг будут реализовывать новые сервисы, необходимо будет оценить их и определить правильный уровень данных и рабочих нагрузок, для которых они подходят. Правительства и органы надзора будут продолжать обновлять или выпускать новые правовые нормы, соблюдая которые, организации смогут хранить данные своих клиентов в безопасности.

В этом контексте обеспечение качества — это процесс постоянной оценки и тестирования систем для гарантии соответствия стандарту, определенному при постановке бизнес-задач. В традиционном смысле этот процесс гарантирует, что код функционально корректен, не вызывает проблем с безопасностью, не имеет других недостатков и соответствует прочим нефункциональным требованиям. Аналогичным образом все устроено в облаке. Однако к нефункциональным требованиям теперь относятся и собственные свойства облака. Например, оптимизация затрат в облаке имеет решающее значение для предотвращения разрастания облаков, и в ходе обеспечения качества следует проверять, являются ли сервисы облачных поставщиков подходящими и правильно ли они реализованы. Процесс обеспечения качества затрагивает не только код и сервисы, но и конвейер развертывания, доступность системы и даже последствия непредвиденных изменений распределенной архитектуры. Как уже говорилось, обеспечение качества должно быть внедрено в процессы управления проектами компании, чтобы проекты разрабатывались с учетом этих принципов еще до развертывания в реальных условиях.

Другой необходимый системам критический компонент — это возможность аудита. Независимо от того, находится ли система в зоне ответственности внешнего или только внутреннего аудитора, данный процесс обеспечивает подотчетность и отслеживаемость для выявления дефектов и проблем с безопасностью. Со временем даже самые отличные системы отходят от своей первоначальной архитектуры и стратегии безопасности. Это естественно и ожидаемо, так как условия бизнеса меняются и внедряется новая функциональность. Важно, чтобы процесс аудита

управления, безопасности и соответствия требованиям оставался неизменяемым или улучшался вместе с системой.

Развертывание новых бизнес-функций не означает, что уровень безопасности должен меняться, поэтому непрерывный аудит периметра безопасности гарантирует, что отклонение от исходного состояния не будет чрезмерным. У поставщиков облачных сервисов обычно есть сервисы конфигурации, которые периодически анализируют общий ландшафт облака и сохраняют его в цифровом формате, идеально подходящем для аудита и сравнительного тестирования. Эти сервисы конфигурации можно расширить, чтобы не только получить представление об облачной среде, но и выполнить пользовательские проверки на уровне системы и сохранить выходные данные вместе с облачной конфигурацией. Благодаря автоматизации аудиторские проверки могут выполняться через короткие промежутки времени, а их результаты сравниваться программным образом, чтобы гарантировать, что в среду не было внесено ничего вызывающего отклонение от требуемого состояния. Автоматизированный аудит — способ обеспечения соответствия требованиям для развитых облачных организаций.

Хотя обеспечение качества и аудит не являются частью *модели зрелости cloud native (CNMM)*, они затрагивают все три оси. Компании, внедрившие модель зрелости CNMM, постоянно следят за появлением новых и развитием существующих облачных сервисов, способных ускорить и удешевить решение бизнес-задач, не выходя за рамки допустимых рисков и требований к соблюдению правовых норм. Они автоматизируют все, включая проверки соответствия, аудиторские действия и сквозные процессы развертывания и обеспечения качества, так что дрейф системы не вызывает проблем с безопасностью. Они направляют свои усилия на то, чтобы убедиться, что их приложения соответствуют лучшим практикам облачных вычислений, и автоматизируют проверку кода на наличие уязвимостей, пробелов в безопасности, неэффективности затрат и условий появления непредвиденных изменений.

Базовые облачные операционные платформы и понятные отправные точки

Когда компании начинают переход в облако, они часто не знают, с чего начать, чтобы обеспечить безопасную базовую среду. Даже если все люди, процессы и технологии находятся в гармонии, отсутствие понятной отправной точки может привести к замедлению или даже к провалу на ранних стадиях деятельности, связанной с облаком. Процесс подготовки базовой среды для будущих облачных проектов имеет множество различных нюансов, часть из которых будет рассмотрена в этом разделе. Мы также поговорим о том, как выбрать правильный подход для отдельной организации.

Как правило, хорошей отправной точкой является утверждение технологического аспекта и определение правильной конфигурации учетных записей, виртуальных

сетей, состояния безопасности, ведения журналов и других базовых решений. Таким образом можно гарантировать, что базовая отправная точка соответствует требованиям компании, имеет достаточно возможностей для роста и может быть разработана с учетом существующих у персонала навыков работы в облаке. В некоторых отраслях наличие одной или нескольких дополнительных моделей управления для обеспечения надлежащего сопоставления внешних элементов управления с облачной средой позволит соответствующим органам аудита проводить аудиторские проверки. Эти концепции более подробно рассматриваются в следующих разделах.

Отправные точки в облаке

Отправные точки в облаке — это все технические и эксплуатационные аспекты, которые необходимо утвердить и спроектировать до фактического развертывания любых рабочих процессов. Существует ряд областей, относящихся к данной категории, но не все они необходимы для успешного перехода в облако. Однако желательно принять их все во внимание. На высоком уровне эти аспекты заключаются в следующем.

- Разработка структуры аккаунта, выставление счетов и маркировка.
- Требования к проектированию сети и межсетевому взаимодействию.
- Определение центральных общих сервисов.
- Обеспечение безопасности и соответствия требованиям аудита, ведение журнала.
- Обеспечение автоматизации и внедрение IaC.

Разработка структуры аккаунта

Стратегия разработки структуры аккаунта — это начало начал, и в каждой компании будут собственные требования к настройке учетных записей. Небольшим компаниям, у которых объем рабочих нагрузок в облаке невелик, может быть достаточно и одной учетной записи. По мере роста размера и сложности организации, как правило, появляется множество отдельных учетных записей, которые позволяют обрабатывать счета, уменьшать последствия непредвиденных изменений, разделять обязанности и требования к окружению или управлению. Использование иерархической структуры, в которой есть основная учетная запись и множество дополнительных, — распространенное явление. Такая структура обеспечивает разграничение задач между учетными записями, но при этом сохраняет возможность объединения определенных компонентов, включая систему безопасности, сетевую инфраструктуру и среду выполнения.

Часто главный аккаунт используется исключительно для выставления счетов и агрегирования затрат. Это позволяет создать единый интерфейс для выставления счетов, а затем разделить его на вспомогательные аккаунты для возврата средств

или просто для понимания того, какие операции являются самыми затратными. Такая система часто позволяет оптимизировать расходы с учетом разных составляющих тарифа, поддерживая оптовые скидки или распределение зарезервированных ресурсов. Вспомогательные или дочерние аккаунты обычно формируются с учетом индивидуальных требований. Независимо от запросов компании, важно понять стратегию облачного аккаунта на самом раннем этапе, чтобы обеспечить его согласование с потребностями всех заинтересованных сторон.

Разработка сети

Следующим после создания учетных записей важным компонентом становится проектирование сети. Учетные записи в основном представляют собой административные конструкции, а возможность развертывания рабочих нагрузок и защиты данных обеспечивается на уровне архитектуры сети. У большинства поставщиков облачных услуг есть концепция виртуальной сети, которая позволяет индивидуально проектировать сетевое пространство, необходимое для удовлетворения различных требований. Затем эти виртуальные сети могут быть дополнительно разделены на подсети, которые учитывают требования к потоку сетевого трафика, например обеспечивают трафик с внешней маршрутизацией (другими словами, общедоступная подсеть) или только внутренний (частные подсети). Подобно разработке учетной записи, проектирование виртуальной сети имеет решающее значение, так как после создания ее трудно изменить, поэтому следует учитывать потребности роста и рабочей нагрузки.

Как правило, одной учетной записи может принадлежать много виртуальных сетей, и они могут взаимодействовать друг с другом или даже с виртуальными сетями в других учетных записях. Таким образом, они являются дополнительным средством уменьшить последствия непредвиденных изменений и изолировать рабочие нагрузки или данные.

Центральные общие сервисы

После того как вы определили и связали между собой учетные записи и виртуальные сети, следующим объектом пристального внимания должна стать работа централизованных общих сервисов. Некоторые сервисы будут стандартными во всем облачном пространстве, и попытка реплицировать их в разные учетные записи или виртуальные сети контрпродуктивна и даже рискованна. В большинстве случаев компаниям с несколькими учетными записями и виртуальными сетями следует использовать общую учетную запись или виртуальную сеть для хранения всех этих сервисов. Это можно рассматривать как виртуальный концентратор и архитектуру «звезда», в которой каждая виртуальная сеть вновь подключается к концентратору, на котором развернут один экземпляр сервиса. Хорошие примеры централизованных общих сервисов — это службы хранения журналов, службы каталогов, сервисы проверки подлинности, CMDB или каталоги сервисов и инструменты мониторинга.

Требования к безопасности и аудиту

Требования к безопасности и аудиту в конечном счете должны распространяться на все облачное пространство. Концепции учетных записей, сетей и разделяемых сервисов будут целиком опираться на стратегию разработки средств обеспечения безопасности и аудита. Как обсуждалось в пункте «Центральные общие сервисы», сервисы каталогов и проверки подлинности важны для защиты доступа к рабочим нагрузкам, данным и учетным записям. Кроме того, критически важными аспектами безопасности и аудита являются надежная структура ведения журналов и процесс аудита конфигурации.

Существует множество разных журналов, доступных в облачной учетной записи для рабочих нагрузок, одни из них предоставляются поставщиком облака, а другие встроены в отдельные приложения. Чем больше журналов собрано, тем лучшие решения можно принять по всем аспектам рабочей нагрузки. Наличие центральной структуры ведения журнала позволяет инстансам, контейнерам, сервису платформы или другому компоненту рабочей нагрузки действовать как машина, не создающая проблем. Объединение журналов рабочей нагрузки с другими, такими как сетевые потоки, вызовы API, задержка пакетов и др., позволяет понять, как ведет себя общая облачная среда, и дает возможность в будущем воспроизвести события посредством агрегирования журналов. Этот мощный метод позволяет лучше ориентироваться в рабочих нагрузках и обеспечивать их безопасность по мере роста и усложнения облачного хранилища.

Возможность выполнения аудита — еще один важный компонент, который необходимо рассмотреть. Как обсуждалось ранее, существуют сервисы облачных поставщиков, которые сохраняют полезные во многих отношениях моментальные снимки или представления общей конфигурации среды через определенные промежутки времени. Во-первых, это позволяет проводить последовательные аудиторские проверки, чтобы доказать, что облачная среда соответствует согласованным требованиям управления. Во-вторых, это дает возможность администраторам облачных вычислений проверять дрейф среды, неверные конфигурации, злонамеренные изменения конфигурации и даже оптимизацию затрат.

Автоматизация и IaC являются ключевыми факторами, которые делают возможными развертывание такого рода систем ведения журнала и аудита, и они напрямую связаны с моделью зрелости облака, которую компания хочет повысить по мере взросления в плане перехода в облако.

Руководство по внешнему управлению

Существует множество различных принципов управления, разработанных для поддержки безопасных и совместимых операционных сред в облаке. В зависимости от отраслевой иерархии, требований к критически важным данным или требований внешнего соответствия у компаний есть варианты улучшения базовой отправной точки в облаке. Даже для организаций, которые не относятся к определенной от-

расли или не подпадают под требования соответствия, согласование одного или нескольких из руководящих принципов управления обеспечит строгое соблюдение международных требований к аппаратной защите облачных ресурсов и поможет в процессах внутреннего аудита сопоставления элементов управления. Далее приведены некоторые важные принципы руководства и модели управления для развертывания совместимых рабочих нагрузок в облаке.

Национальный институт стандартов и технологий (NIST)

Существует множество различных платформ NIST, готовых к применению. Amazon Web Services проделала большую работу по созданию справочных документов, руководств для быстрого старта и процедур ускоренного развертывания для самых популярных из них:

- NIST SP 800-53 (редакция 4) (https://www.wbdg.org/files/pdfs/dod_cloudcomputing.pdf);
- NIST SP 800-171 (http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-171.pdf);
- OMB TIC Initiative — FedRAMP Overlay (pilot) (https://www.fedramp.gov/draft-fedramp-tic-overlay/);
- DoD Cloud Computing SRG (https://www.wbdg.org/files/pdfs/dod_cloudcomputing.pdf) (https://aws.amazon.com/about-aws/whats-new/2016/01/nist800-53-standardized-architecture-on-the-aws-cloud-quick-start-referencedeployment/).

Используя руководства и процедуры ускоренного развертывания, описанные AWS, организация может развернуть стандартизированную среду, приносящую пользу проектам, на которые направлены эти публикации.

Стандарт безопасности данных индустрии платежных карт (PCI DSS)

Тем, кому необходимо обрабатывать данные кредитных карт для своих клиентов, стоит придерживаться рекомендуемого стандарта соответствия PCI DSS (https://www.pcisecuritystandards.org/). Чтобы помочь клиентам настроить среду, совместимую с PCI DSS, Amazon Web Services разместила подробную эталонную архитектуру и сопоставление элементов управления по адресу https://aws.amazon.com/quickstart/architecture/accelerator-pci/.

Закон об ответственности и переносе данных о страховании здоровья граждан (HIPAA)

Клиенты, которые хотят выполнять приложения для работы с конфиденциальными данными, регулируемые актом HIPAA, и планируют хранить защищенную медицинскую информацию (PHI), должны понимать, каковы будут последствия, и иметь право действовать от имени конкретного поставщика облачных услуг. Не все услуги облачных поставщиков входят в сферу действия соглашения о бизнес-партнерстве HIPAA, и заказчик сам определяет тип данных и рабочую нагрузку, которую они проектируют. Дополнительную информацию о том, как этого

добиться в AWS, можно найти по адресу https://d0.awsstatic.com/whitepapers/compliance/AWS_HIPAA_Compliance_Whitepaper.pdf.

Центр интернет-безопасности (CIS)

CIS (https://www.cisecurity.org/cis-benchmarks/) предлагает процедуру согласованного анализа с участием специалистов в соответствующей предметной области для определения рекомендаций по развертыванию и настройке сервисов с целью удовлетворения требований безопасности для широкого круга элементов управления. Полное объяснение того, как этого добиться в Amazon Web Services, можно найти по адресу https://d0.awsstatic.com/whitepapers/compliance/AWS_CIS_Foundations_Benchmark.pdf.

Миграция в облако в сравнении с разработкой с нуля

После того как решение о переходе в облако принято, определены структуры и ограничения и достигнут некоторый первоначальный успех, возникает вопрос о переносе большого количества рабочих нагрузок. Миграция в облако определяется как перемещение приложений, данных или других компонентов из существующего местоположения (обычно локального) в облако. Проект разработки с нуля — это проект, в котором нет никаких ограничений, накладываемых на разработку, следовательно, результатом является совершенно новая реализация. Миграции и разработки с нуля часто существуют бок о бок: имеющиеся рабочие нагрузки переносят в целевую облачную операционную среду, а новые проекты разрабатывают уже как полностью облачные.

В этом разделе будут рассмотрены общие шаблоны миграции, инструменты, которые при этом используются, а также то, как «чистая» облачная разработка вписывается в историю миграции. В этой книге вы не найдете подробностей о миграции в облако. Однако миграции являются важной частью процесса архитектуры cloud native, поэтому важно понимать, где они пересекаются.

Шаблоны миграции

Миграция в облако часто происходит с использованием модели 6R, описывающей шесть способов: *Rehost* (рехостинг), *Replatform* (смена платформы), *Repurchase* (повторная покупка), *Refactor* (рефакторинг), *Retire* (списание) и *Retain* (сохранение). Каждая из этих шести R занимает свое место в обсуждении перехода в облако. Часто оказывается, что проще или дешевле сохранить существующее приложение *(Retain)* или вообще отказаться от его применения *(Retire)*, чем пытаться перевести его в облако. Переход на аналогичное облачное приложение или повторная покуп-

ка *(Repurchase)* могут иметь последствия для всей корпоративной архитектуры, особенно когда решено заменить существующую рабочую нагрузку предложением *«программное обеспечение как сервис» (SaaS)*, которое является облачным, но полностью управляется другой компанией.

Поэтому удаление, сохранение и повторная покупка не будут обсуждаться здесь в разрезе cloud native. Оставшаяся часть раздела будет посвящена рехостингу, смене платформы и рефакторингу.

Рехостинг

Рехостинг, который часто описывают как «поднять и перенести» (lift-and-shift), — это самая быстрая возможность перейти в облако, в чистом виде он подразумевает перенос всей вашей среды в максимально неизменяемом виде в облачную инфраструктуру, которая действует так же, как существующие системы. Этот шаблон миграции часто выбирают потому, что он позволяет выполнять довольно небольшие объемы работ по предварительному анализу и обеспечивает быструю окупаемость инвестиций. На рынке существует множество инструментов, которые поддерживают данный тип миграции, выполняя репликацию на уровне блоков или упаковку образов, чтобы доставить экземпляры в облако. Кроме того, имеются инструменты, позволяющие серверам баз данных переносить свои данные, включая их структуру, аналогичным образом. Недостаток этого типа миграции заключается в том, что, как правило, из-за выигрыша в скорости не реализуются реальные преимущества облака. Часто миграция в облако выполняется из-за неотвратимого события (например, окончания срока аренды центра обработки данных), и скорость становится наиболее важным требованием. Существует также мнение, что рехостинг в облаке ускорит возможность смены платформы или рефакторинга приложений в облаке, что снизит риск внесения изменений во время миграции, как показано на рис. 2.1.

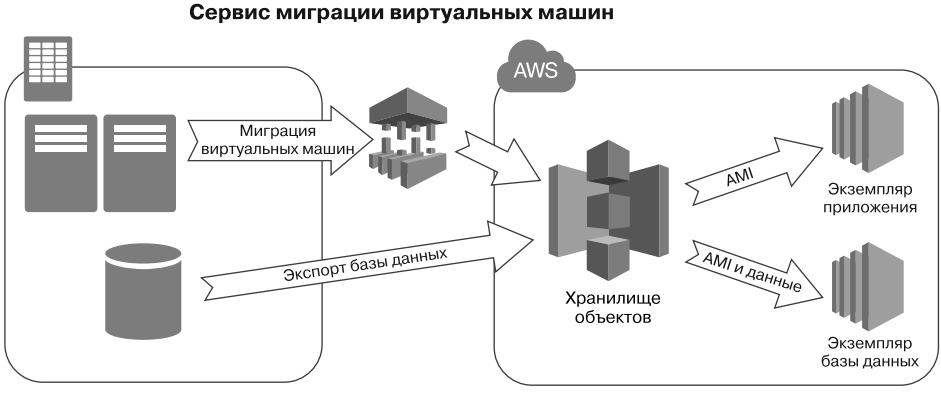


Рис. 2.1

Смена платформы

Смена платформы — это, по сути, более надежная и сложная версия рехостинга, при которой компоненты существующих приложений переносятся в облачные аналоги. В процессе миграции в архитектуру рабочей нагрузки вносят изменения, позволяющие использовать преимущества поставщиков сервисов cloud native, например перемещать приложение точно так же, как есть, но с переводом локальной базы данных в совместимую облачную службу базы данных, управляемую облачным поставщиком. Хотя это изменение кажется незначительным, оно вносит изменения в архитектуру рабочей нагрузки и должно проводиться как мини-рефакторинг с тестированием и документацией. Смена платформы — это тоже шаблон, который можно выполнить во время миграции из локальной среды в облако или уже после переноса приложения в облако, как показано на рис. 2.2.

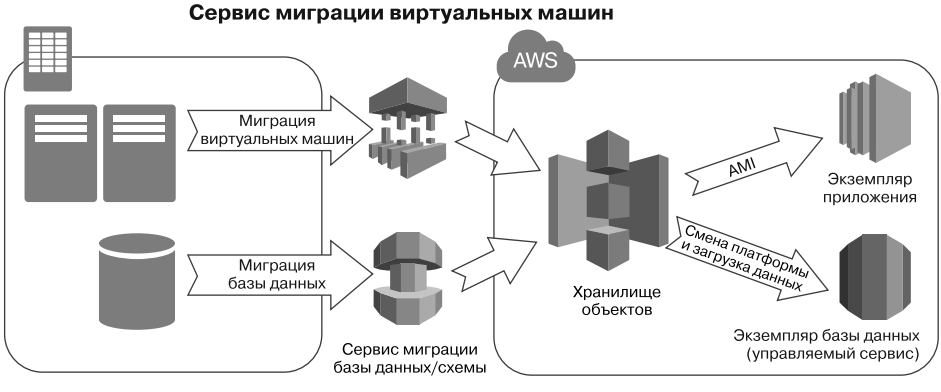


Рис. 2.2

Рефакторинг

Рефакторинг (изменение архитектуры) — это переосмысление архитектуры и разработки приложения, обычно с использованием облачных функций. Этот тип шаблона миграции, по сути, представляет собой переписывание приложения, чтобы привести его в соответствие с CNMM (обсуждалась в главе 1), добавить автоматизацию и спроектировать приложение для создания архитектуры cloud native. В то время как рехостинг и смена платформы больше соответствуют тому, что обычно называют миграцией, рефакторинг приложения в процессе миграции становится все более популярным, поскольку компании осознают, что простой перенос приложения в облако не позволит им реализовать все преимущества своего бизнес-кейса. Рефакторинг по-прежнему применяется к миграциям, поскольку в большинстве случаев данные преобразуются в более новый механизм базы данных (другими словами, NoSQL), приложение перестраивается для разделения компонентов сервисов (или микросервисов), и, что наиболее важно, бизнес-логика остается неизменяемой (или улучшается) в течение этого процесса, как показано на рис. 2.3.

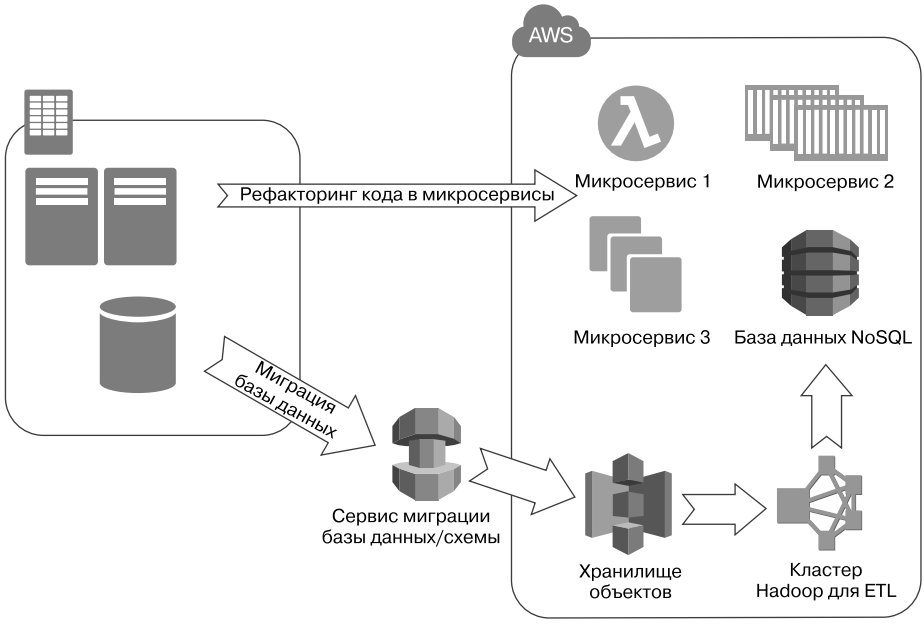


Рис. 2.3

Миграция или разработка с нуля?

В этой главе обсуждались стимулы и соглашения для перехода в облако, и мы подробнее рассмотрим, как предприятие должно переносить туда существующие рабочие нагрузки. Хотя некоторые из рассмотренных подходов предусматривают этапы разработки, посвященные рефакторингу или смене платформы приложения, они все еще являются шаблонами миграции. Итак, вопрос заключается в следующем: когда организация использует разработку с нуля вместо миграции? Как правило, когда компания решается стать облачной, она сосредоточивает значительные усилия на переносе существующих рабочих нагрузок в облако. В то же время все свои новые разработки она будет выполнять уже в облачной среде, по сути, с нуля. Обычно для нового проекта или рабочей нагрузки по-прежнему требуются точки взаимодействия с устаревшими приложениями независимо от того, находятся ли они в облаке или все еще переносятся, поэтому общим для этих двух процессов является интеграция точек взаимодействия.

По мере того как миграционные процессы ослабевают, приоритетом становится концентрация на новых разработках в облаке. Вслед за миграцией в организации начинается захватывающий период, характеризующийся уверенным владением облачными технологиями и оптимизмом в отношении того, каким образом облако может изменить компанию. Будучи проектировщиком облачных систем на этом этапе, вы получите возможность предлагать смелые идеи и избавляться от старых преград.

Резюме

В этой главе мы определили причину, по которой компании принимают решение перейти в облако. Понимание ключевых факторов — гибкости, оптимизации затрат, безопасности, управления, расширения, привлечения талантов и внедрения инноваций, а также уровня их зрелости в компании — позволит принять окончательное решение о переходе. После этого очень важно определить и внедрить операционную модель, благодаря которой заинтересованные стороны, а также руководители проектов и процессов внесения изменений могут быть уверены в том, что требования к рискам соблюдаются и находятся под контролем, обеспечивая успешнную работу в облаке. Наконец, мы рассмотрели основные шаблоны миграции, которые позволяют переносить рабочие нагрузки в облако. Хотя некоторые организации пропустят миграцию и начнут работу в облаке с нуля, большинству компаний потребуется этот шаг, и важно, чтобы они сделали его правильно. Компания, которая исповедует политику «облако превыше всего», перенесет существующие рабочие нагрузки, но все новые приложения создаст в облаке в соответствии с собственными шаблонами архитектуры cloud native.

В следующей главе рассмотрим требования на уровне приложений для архитектур cloud native, включая жизненный цикл разработки системы, сервисно-ориентированные архитектуры (SOA), микросервисы и бессерверные вычисления. Мы подробно изучим различные фреймворки и методологии и поймем, как они вписываются в принципы проектирования cloud native.

3 Разработка приложений cloud native

В данной главе мы более подробно изучим разработку архитектур cloud native, использующих микросервисы и бессерверные вычисления для реализации модели зрелости CNMM. В рамках эволюции компьютерных систем от монолитной до гораздо более совершенных архитектур в этой главе рассмотрим контейнеры, оркестрацию и бессерверные решения, а также то, как они сочетаются друг с другом и почему считаются развитыми на оси проектирования, ориентированного на приложения.

Монолитные системы, микросервисы и все, что между ними

Приложения, построенные по типу «клиент — сервер», всегда были популярными. Однако по мере развития сетевых технологий и шаблонов проектирования необходимость в менее тесно связанных приложениях определила появление *ориентированных на сервисы архитектур (SOA)*. SOA — это концепция разделения компонентов, составляющих монолит или сервер, на множество дискретных бизнес-сервисов. Компоненты SOA по-прежнему являются автономными. Однако они значительно меньше по объему, чем традиционные монолитные приложения, и обеспечивают более быстрое обслуживание и разобщенные взаимодействия. Традиционный клиент все еще может рассматриваться как компонент приложения SOA, но вместо прямой связи с монолитным сервером появится промежуточный уровень или серверная шина, которая принимает вызов и передает его другим сервисам для обработки. Эти сервисы могут предлагать сохранение данных или собирать дополнительную информацию для принятия бизнес-решения. Подход SOA дал возможность распределять системы и позволил существенно изменить взаимодействие между различными системами, даже теми, которые находятся за пределами внутренней инфраструктуры компании.

Эти подходы продолжают развиваться в рамках облачных вычислений благодаря постепенному удешевлению данных услуг, а также все более доступным ресурсам для создания сервисов. Микросервисы, похожие на SOA, представляют собой

эволюцию, которая позволяет разбить конкретный сервис на еще более мелкие компоненты. Сервис в SOA — это черный ящик, который целиком выполняет бизнес-функцию. Каждый из микросервисов предназначен для выполнения какой-то части этой общей бизнес-функции. Это улучшает поддержку, увеличивает скорость обновления функциональности системы и уменьшает последствия непредвиденных изменений кода или аппаратных сбоев. Архитектура cloud native, как определено в главе 1, будет иметь разные уровни зрелости (в соответствии с моделью зрелости облачно-ориентированных систем), предусматривающие, к примеру, использование облачно-ориентированных сервисов, разработку, ориентированную на приложения, автоматизацию и принципы проектирования приложений. Микросервисы предоставляют способ соблюдения всех этих принципов, чтобы обеспечить масштабируемость и отказоустойчивость, которых трудно достичь с помощью любого другого традиционного метода проектирования системы.

Шаблоны проектирования системы

В ходе эволюции проектирования систем различные факторы влияли на то, как создавались и разворачивались системы для решения все более сложных бизнес-задач. В данном разделе более подробно будут рассмотрены шаблоны проектирования, дано дополнительное предоставление о том, как они работают, и о трудностях, стимулировавших их развитие. Чтобы понять, как развивалась архитектура решений, важно определить несколько ключевых понятий, а именно базовые компоненты (примитивы), подсистемы и системы.

Независимо от шаблона проектирования примитивы являются базовым уровнем общего решения, в зависимости от стиля их часто называют функциями, задачами или микросервисами. Предполагается, что примитив — это основная единица, способная выполнять действие. Примитивы часто настолько специализированы, что не могут непосредственно влиять на подсистему, к которой принадлежат. Вместо этого они быстро и эффективно выполняют одну задачу. Подсистемы — это логические группы примитивов, которые формируют дискретную бизнес-функцию. Подсистемы не обязательно должны быть одним компонентом, хотя могут зависеть от архитектуры проекта. Однако им нужен логический поток, который приводит к реализации бизнес-функции. Примитивы, составляющие подсистему, могут быть вызовами функций в одном и том же блоке кода или в отдельном микросервисе, вплоть до получения конечных результатов в системе. Системы являются верхним уровнем решения и часто состоят из множества подсистем, которые взаимодействуют друг с другом или с другими системами для прохождения всего процесса от начала до конца.

Архитектуры cloud native являются продуктом такой эволюции. Важно понимать структуру архитектуры и происхождение шаблонов, чтобы процесс развития мог продолжаться и мы могли использовать и постепенно развивать архитектуру, которая лучше всего соответствует определеным бизнес-требованиям. Есть три

основных шаблона проектирования: монолитный, «клиент — сервер» и с использованием сервисов. Все они работают на базе систем, подсистем и примитивов. Однако реализация каждого имеет ключевкю особенность, которая позволяет им быть более надежными.

Монолитный

С первых дней существования информационных технологий и появления вычислительных ресурсов для ускорения вычислительных задач предпочтительным шаблоном проектирования для приложения был монолитный. Проще говоря, монолитный шаблон проектирования — это шаблон, в котором все аспекты системы автономны и независимы от других систем или процессов. Этот шаблон прекрасно работал с большими монолитами мейнфреймов, которые позволяли размещать логику кода, устройства хранения и вывода в одном месте. Сетевое взаимодействие, каким мы его знаем сегодня, было невозможно, а доступ к небольшому количеству соединений можно было получить значительно медленнее, что не позволяло подключить дополнительные системы. Когда определялись новые бизнес-требования, они разрабатывались непосредственно в существующем монолите, что делало их критическими и очень тесно связанными. В конечном счете монолитные шаблоны проектирования позволяли выполнить все задачи и достичь желаемого результата.

По мере развития технологий и удовлетворения потребностей в вычислительных ресурсах разработчики систем обнаружили, что добавление новых функций в монолит становится обременительным и может привести к появлению трудно распознаваемых ошибок. Они также поняли, что повторное использование компонентов и задач — это отличный способ сэкономить время, деньги и уменьшить количество ошибок при изменении кода. Модульный подход к проектированию систем, который по-прежнему был в основном монолитным, позволял реализовывать повторное применение и более целенаправленные технические обновления без ущерба для приложения или системы в целом. Эта философия продолжала развиваться по мере совершенствования технологий и позволяла разделять некоторые из крупнейших монолитов на отдельные дискретные подсистемы, которые затем можно было обновлять независимо. Концепция разделения компонентов — основной движущий фактор в эволюции системных архитектур, и по мере развития технологий это разделение будет только усиливаться.

Клиент — сервер

В конце концов, когда затраты на технологии снизились, а шаблоны проектирования стали гораздо сложнее, стал популярен новый стиль архитектуры — приложения «клиент — сервер». Монолит все еще использовался для обеспечения необходимой вычислительной мощности и хранения данных. Однако достижения в области сетей и новые концепции баз данных позволили для обеспечения взаимодействия с пользователем и передачи данных на сервер применять внешнее приложение, или клиент.

Клиентское приложение все еще было тесно связано с серверным. Тем не менее такая архитектура позволяет многим клиентам подключаться к серверу и, следовательно, распространять приложение среди сотрудников на местах или обычных интернет-пользователей. Эта эволюция модульного подхода также изменила способ, которым ИТ-отделы развертывали и обслуживали системы. В частности, за клиентские и серверные компоненты отвечали разные группы сотрудников, а профессиональные навыки начали смещаться в сторону дальнейшего повышения эффективности.

Сервисы

Клиент-серверные приложения работали очень хорошо, но связь между компонентами замедляла и делала рискованным развертывание критически важных приложений. Разработчики систем продолжали искать способы отделения компонентов друг от друга, уменьшения последствий непредвиденных изменений, увеличения скорости развертывания и снижения рисков. В шаблоне проектирования сервисов бизнес-функции разбиты по конкретным задачам и выполняются автономно, что не требует знаний или зависимости от состояния другого сервиса. Автономное независимое состояние сервиса само по себе является определением разделения и представляет собой серьезное эволюционное изменение по сравнению с монолитной архитектурой. Изменение в архитектуре сервисов напрямую связано со снижением стоимости вычислений и хранилищ, а также с повышением уровня сложности сетей и методов обеспечения безопасности. Снижение затрат позволяет создавать множество вычислительных экземпляров сервисов в более широких географических регионах и с более разнообразным набором функциональных требований бизнеса.

Сервисы — это не просто пользовательский код, разработанный для решения бизнес-задач. На самом деле в процессе совершенствования своих сервисов поставщики облачных услуг нередко создают управляемые решения, которые помогают устранить недифференцированную рутинную работу при эксплуатации некоторой части системы. Эти отдельные сервисы относятся к категории услуг и зачастую составляют конкретные функции в большом распределенном изолированном решении, например, с применением сервиса очередей или сервиса хранения в качестве компонентов архитектуры. В системе с монолитной архитектурой все эти компоненты пришлось бы разрабатывать и использовать в рамках единого стека приложений с жесткими взаимными связями и существенным риском сбоев, ухудшения производительности или написания дефектного кода. Продвинутые облачные архитектуры изначально проектируются для использования этих сервисов, предоставляемых поставщиком, в дополнение к собственным функциям; все вместе это работает как единое целое, реагируя на разные события и в конечном итоге помогая решать бизнес-задачи настолько просто, насколько это возможно.

В то время как архитектуры на основе сервисов становились очень популярны в качестве архитектур cloud native, для проектирования сервисов потребовалось время.

Начиная с первого поколения сервис-ориентированных архитектур и заканчивая микросервисами и функциями, каждый следующий подход направлен на дальнейшее дробление функциональности с целью как можно более жесткого разделения взаимодействующих компонентов.

Ориентированные на сервисы архитектуры

Шаблон проектирования сервисов, существовавший до появления облака, — это ориентированные на сервисы архитектуры (SOA). Он является первой крупной эволюцией разделения монолитного приложения на отдельные части для уменьшения последствий непредвиденных изменений и ускорения процессов разработки и развертывания приложений в целях поддержки скорости бизнеса. Шаблоны проектирования SOA по-прежнему зачастую представляют собой большие приложения, разделенные на подсистемы, которые взаимодействуют друг с другом так же, как в монолитной системе. Однако между ними часто имеется промежуточный уровень, например *сервисная шина предприятия (ESB)*.

Использование ESB для передачи запросов на обслуживание делает возможным применение разнородных протоколов связи и дальнейшее разделение сервисов — зачастую посредством обмена сообщениями. ESB допускает трансляцию сообщений, маршрутизацию и другие виды промежуточных механизмов, чтобы обеспечить получение потребительским сервисом сообщений в правильной форме в нужное время. Поскольку шаблоны проектирования SOA часто объединяются в большую и сложную корпоративную систему, отдельные сервисы нередко содержат полный набор бизнес-логики для конкретной подсистемы. Такая структура позволяет частично обособить сервис и отправлять сообщения через EBS для предоставления или запроса данных из других подсистем.

Данный подход к разработке SOA значительно расширил возможности систем быть более распределенными, быстрее обслуживать потребности бизнеса и использовать преимущества более дешевых вычислительных ресурсов. Однако у SOA есть и недостатки. Поскольку сервисы взаимодействуют с другими компонентами через сервисную шину предприятия, она может быстро стать узким местом или единственной точкой отказа. Поток сообщений становится очень сложным, и любые неполадки в работе ESB могут привести к значительному замедлению или созданию очереди сообщений, что способно нарушить работу всей системы. Кроме того, поскольку для каждого сервиса нужен определенный тип сообщения, отформатированного в соответствии с конкретными требованиями, тестирование всего составного приложения становится очень сложным и неизолированным, что замедляет скорость бизнеса. Например, когда существенно обновляются два сервиса, каждый из которых начинает использовать новые, усовершенствованные форматы сообщений, развертывание потребует обновления как для подсистем, так и для ESB, что может нарушить работу всего приложения, которое они составляют.

Микросервисы

Эволюция SOA привела к дальнейшему разделению функциональности бизнеса на еще более мелкие компоненты, часто называемые микросервисами. Они все еще являются сервисами, как было определено ранее, но сфера их применения и способы взаимодействия изменились, чтобы можно было использовать преимущества технологий и ценообразования. Микросервисы часто используют, чтобы уменьшить часть проблем, с которыми сталкиваются ориентированные на сервисы архитектуры, при этом продолжая совершенствовать модель сервисов. Между архитектурами микросервисов и SOA существует два ключевых различия: объем услуг и методы соединения.

Микросервисы предназначены для дальнейшей специализации сферы услуг. С SOA сервис будет действовать как полноценная подсистема, имеющая большой блок бизнес-функций, в то время как архитектура микросервисов будет разбивать эту подсистему на компоненты, более близкие к примитивам, выполняющим подмножество функций основного сервиса. Микросервис может хранить бизнес-логику или просто выполнять служебные задачи, такие как транзакция базы данных, аудит или ведение журнала. Если он содержит бизнес-логику, его охват будет намного у́же по сравнению с сервисом SOA, и он, как правило, не будет самостоятельным компонентом, способным выполнить бизнес-задачу целиком. Эта разбивка функциональности позволяет сервису масштабироваться независимо от других компонентов и часто выполняется на основе события, чтобы сократить время использования облачных ресурсов и уменьшить плату за них.

Другой ключевой особенностью микросервиса является метод связи. SOA часто связывается с разнородными протоколами через ESB, где микросервисы предоставляют API, который может вызываться любым сервисом-потребителем. Такое предоставление доступа к API позволяет разрабатывать сервис на любом языке, на который не будут влиять другие сервисы, их язык разработки или подход. Благодаря тому что слой API является отдельным компонентом, изменения в логике сервиса часто не требуют координации между потребительскими сервисами, и даже когда сам API модифицируется, необходимо уведомлять только сервисы-потребители, что часто можно выполнить посредством динамической идентификации конечной точки API и навигации.

Почему сервисы важны

Ключевой особенностью облака является способ разработки и развертывания приложений, при этом архитектура на основе сервисов наиболее популярна. Сервисы важны, потому что позволяют разрабатывать приложения из все более мелких компонентов, которые имеют более высокую скорость развертывания, — в результате бизнес-идеи могут быстрее выходить на рынок. Кроме того, чем меньше сервис или функция, тем меньше последствия непредвиденных изменений, а это означает:

неисправность или плохой код влияют только на относительно небольшое количество подключенных сервисов. Такое уменьшение масштаба потенциальных последствий делает развертывание кода куда менее рискованным и уменьшает время, необходимое для отката при появлении чего-то плохого (рис. 3.1).

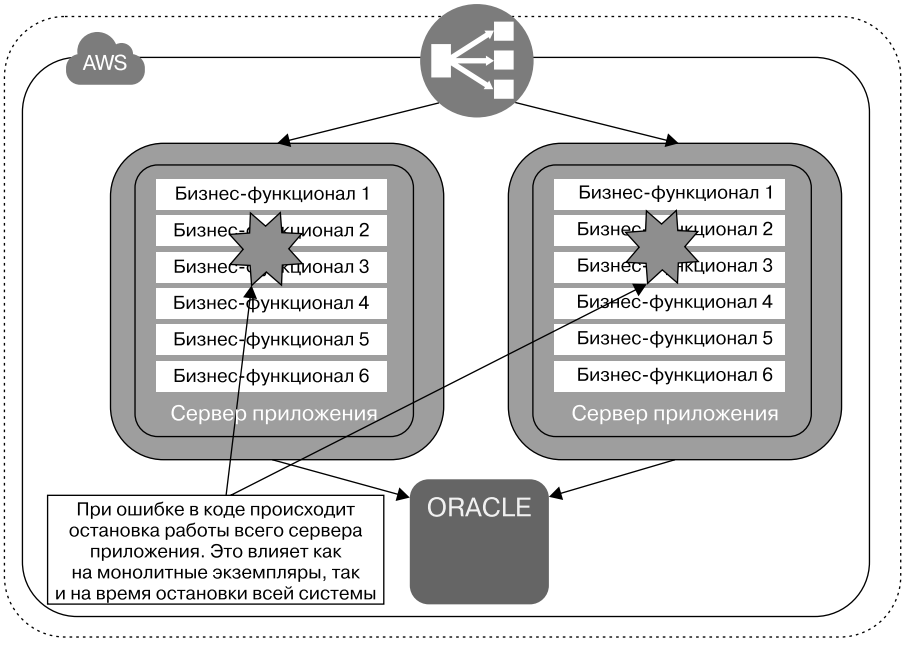


Рис. 3.1

На схеме изображена монолитная система, развернутая в нескольких экземплярах для обеспечения высокой доступности. Однако спроектирована она так, что каждый экземпляр приложения, запущенный на сервере приложений, разворачивается с задействованием полного функционала системы. В этом случае блоки бизнес-функциональности являются примером подсистемы, имеющей много различных функций, или примитивов, которые включены для реализации требуемой функциональности. В данном примере новый код должен выполняться на системном уровне, что может потребовать запланированного простоя и полного повторного развертывания всего приложения. Если в каком-либо фрагменте развернутого кода будет непреднамеренная ошибка в подсистеме или примитиве, это повлияет на все приложение, так как оно развернуто целиком. Незапланированная остановка приведет к откату всего артефакта приложения, даже подсистем, работавших, как предполагалось, в других частях системы. Такой подход не очень гибок и приводит к длительным и напряженным развертываниям способом *«большого взрыва»*, которые должны планироваться на несколько недель или месяцев.

При разделении систем и бизнес-функций на отдельные подсистемы количество точек соприкосновения между этими подсистемами значительно уменьшается, часто они вообще никогда не взаимодействуют. Использование подсистем и примитивов в виде сервисов позволяет добиться того, что вносимые изменения станут гораздо меньше влиять на организацию в целом, в результате чего риск, связанный с этими изменениями, снизится, поскольку последствия затрагивают только сервисы, также непосредственно связанные с ними. В зависимости от требований подсистема может быть изолированным компонентом или разбиваться на множество примитивов.

На рис. 3.2 исходное монолитное приложение было перепроектировано и развернуто в отдельных подсистемах и примитивах с помощью набора инстансов, контейнеров, функций и сервисов поставщиков облачных вычислений. Не все сервисы взаимодействуют друг с другом или сохраняют данные, поэтому при наличии проблем в среде большинство из них все еще можно использовать. Подсистема *1* изображена как имеющая сбой из-за неправильного развертывания кода, который будет обнаружен и устранен командой обслуживания. Между тем все прочие сервисы по-прежнему работают без изменений, если только не взаимодействуют с подсистемой *1*. На схеме также показано, как влияет отказ одного инстанса из-за аппаратного сбоя — обычный случай, который не должен влиять на всю систему или какой-то отдельный сервис. Подсистема *3* обнаружит потерю инстанса и автоматически развернет новый инстанс или контейнер, чтобы не отставать от входящих запросов. Примитивные микросервисы *5* и *6* выполняются как блоки кода на основе событий, которые взаимодействуют с другими примитивами для сохранения данных, взаимодействия с очередями сообщений или выполнения бизнес-функций.

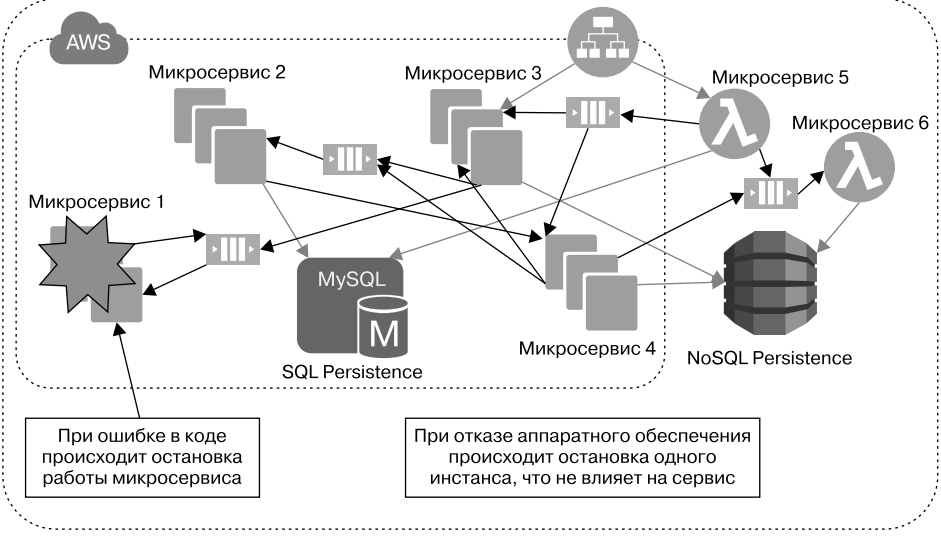


Рис. 3.2

Сервисы важны, так как они позволяют разделить функциональность, уменьшить последствия изменений, ускорить бизнес-процессы и, если все сделано правильно, снизить стоимость работы всей системы.

Контейнеры и бессерверность

Архитектуры cloud native становились все более продвинутыми по мере развития методик проектирования, в которых применяются новейшие достижения облачных технологий. Микросервисы в настоящее время являются важным направлением в развитии архитектур, поскольку обеспечивают массовое разделение компонентов при использовании сервисов cloud native способами, которые были бы невозможны с локальным программным обеспечением. Однако микросервисы — это просто шаблон. Существует несколько способов создания микросервиса с помощью различных технологий и подходов, а контейнеры и бессерверные подходы — наиболее распространенные из них. Нельзя сказать, что микросервисные системы не могут быть разработаны с более традиционными виртуальными инстансами — при правильном применении они все еще вполне пригодны. Это означает, что контейнеры и бессерверные технологии позволяют разрабатывать и развертывать системы в масштабе и с гибкостью, которые обычно обеспечиваются для микросервисов. Они больше соответствуют целям микросервисов. В этом разделе рассмотрим контейнеры и серверы и объясним, что это такое и как их можно использовать для разработки архитектур cloud native.

Контейнеры и оркестрация

Контейнеры стали естественным развитием дополнительной изоляции и виртуализации аппаратного обеспечения. В то время как виртуализация дает возможность запустить на физическом сервере несколько хостов, тем самым обеспечивая большую эффективность и позволяя задействовать ресурсы данного конкретного сервера, контейнеры ведут себя аналогично, но при этом они гораздо легче, лучше переносимы и масштабируемы. Обычно контейнеры работают под управлением операционной системы Linux и иногда Windows. Тем не менее многие излишние и неиспользуемые компоненты удаляются; их сфера применения ограничивается лишь традиционными процедурами запуска, завершения работы и управления задачами. На следующем этапе разработчики могут подключать определенные библиотеки или языки и развертывать код и конфигурации непосредственно в файл контейнера. Затем этот контейнер рассматривается как артефакт развертывания и публикуется в реестре для развертывания или обновления во всем парке контейнеров, уже запущенных в кластере через службу оркестрации.

Благодаря своей легкости контейнеры стали идеальными инструментами для работы в облачной среде. Они обеспечивают бескомпромиссное разбиение приложения

на микросервисы. Масштабирование контейнеров может происходить как за счет максимизации ресурсов на отдельно взятом сервере, так и путем объединения множества серверов в кластер, где их обслуживанием и планированием их работы занимается сервис оркестрации. Для хранения файлов сервис может использовать локальные временные папки файловой системы, а если данные могут понадобиться в дальнейшем, файлы могут сохраняться в постоянно примонтированном томе, доступном всем контейнерам в узле. Чтобы упростить размещение и масштабирование контейнера, отдельные средства оркестрации используют pod-оболочки — группы контейнеров, которые запускаются и масштабируются как единое целое.

Реестры

Реестр контейнеров — это просто общедоступная или частная область для хранения и обслуживания предварительно собранных контейнеров. Существует множество вариантов использования реестров, и все основные поставщики облачных услуг размещают общедоступные версии реестров, которые можно сделать частными для конкретной организации. Кроме того, существует масса общедоступных реестров, таких как Docker Hub, в котором хранятся предварительно созданные файлы-контейнеры, имеющие множество стандартных готовых к применению конфигураций. Любой из этих вариантов рабочий и зависит лишь от потребностей организации. Однако после завершения фазы обучения рекомендуется всегда использовать частные реестры. Помимо того что этот механизм намного безопаснее для образов частных контейнеров, размещаемых в облачной срде, он также обеспечивает непосредственную близость к сервису оркестрации для ускорения работы конвейера развертывания.

Реестры играют важную роль в конвейерах CI/CD, которые будут задействоваться для развертывания приложений. Возьмем для примера простой конвейер, в котором разработчик сохраняет код в выбранном репозитории (в данном случае в Git). Это сохранение инициирует процесс сборки с использованием программы Jenkins, которая будет собирать контейнер, проверять его с помощью предопределенных сценариев и передавать в сервис реестра. При попадании контейнеров в реестр инструмент оркестрации инициирует извлечение контейнера, после чего этот контейнер будет обновлен или развернут. Существует несколько способов реализации окончательного развертывания: либо непрерывное обновление с нулевым временем простоя, либо новое развертывание в соответствии с требованиями конфигурации.

Оркестрация

Оркестрация контейнеров — довольно трудная задача. Если приложение было разработано для использования контейнеров в виде большого монолитного стека, который выполняет все задачи, необходимые для работы подсистемы, то развертывание этих контейнеров становится довольно простым. Однако это противоречит

цели применения контейнеров и микросервисов, которая заключается в разделении компонентов на небольшие сервисы, взаимодействующие друг с другом или с другими сервисами. Поскольку подсистема разбита на множество отдельных примитивных микросервисов, возможность координировать эти сервисы, состоящие из одного или нескольких контейнеров, становится очень сложной. Некоторые сервисы будут относительно небольшими и станут использоваться в редких случаях, а другие будут очень объемными и потребуют строгого масштабирования и высокой доступности конфигураций. Кроме того, развертывание новых контейнеров в кластере имеет решающее значение и обычно должно выполняться с минимальными простоями или без простоев, что делает сервис, реализующий эти обновления, очень ценным.

На рынке существует множество типов инструментов оркестрации контейнеров, и каждый поставщик облачных вычислений предлагает одну или несколько версий, либо организация может настроить и поддерживать собственные инструменты оркестрации в зависимости от своих требований. Один из самых популярных сервисов оркестрации — Kubernetes, который может работать на любой облачной платформе локально или гибридно. Стоит использовать службу оркестрации контейнеров поставщика облачных вычислений вместо стороннего инструмента прежде всего из-за отлаженного взаимодействия контейнеров и других облачных сервисов, предоставляемых поставщиком. Тем не менее даже сейчас поставщики облачных услуг разворачивают собственные управляемые сервисы Kubernetes. Вот некоторые из ключевых концепций развертывания Kubernetes.

- *Kubernetes Master*. Отвечает за поддержание желаемого состояния кластера.
- *Узел Kubernetes*. Узлы — это машины (виртуальные машины, физические серверы и т. д.), на которых выполняются ваши приложения и облачные рабочие процессы. Kubernetes Master контролирует каждый узел, вы редко будете взаимодействовать с ними напрямую.
- *Pod*. Основной строительный блок Kubernete. Самая маленькая и самая простая единица в объектной модели Kubernetes, которую вы создаете или развертываете. Под — это абстрактный объект Kubernetes, представляющий собой группу из одного или нескольких контейнеров приложения и совместно используемых ресурсов для этих контейнеров. Также под представляет собой запущенный в вашем кластере процесс.
- *Сервис*. Абстракция, которая определяет логический набор подов и политик доступа к ним, иногда называемая микросервисом.
- *Контроллеры и наборы реплик*. Обеспечивают одновременную работу заданного количества реплик подов. Другими словами, они удостоверяются, что под или однородный набор подов всегда запущен и доступен.
- *Развертывание*. Предоставляет декларативные обновления подов и наборов реплик.

- *DaemonSet*. Гарантирует, что все (или некоторые) узлы запускают копию модуля. Когда узлы добавляются в кластер, к ним добавляются и модули. Когда узлы удаляются из кластера, то же происходит и с модулями. Удаление DaemonSet очистит созданные этим процессом поды.

Методы использования контейнеров

Контейнеры — это отличный инструмент разработки и реализации собственных облачных архитектур. Контейнеры хорошо вписываются в модель облачной зрелости, описанную в главе 1. В частности, они являются сервисом cloud native от поставщика, обеспечивают экстремальную автоматизацию посредством CICD и отлично подходят для микросервисов благодаря своей легковесности и масштабируемости. Существует несколько ключевых моделей, в которых контейнеры являются центральным компонентом: микросервисы, гибридные и миграционные развертывания приложений, а также инновации в бизнесе за счет гибкости.

Микросервисы с контейнерами

Контейнеры содержат только минимальную операционную систему и предопределенные библиотеки и компоненты, необходимые для выполнения заданной части бизнес-функций. Из-за этой легковесности они часто считаются синонимами микросервисов. Таким образом, приложения независимо от того, разработаны они как новые или являются частью разбитых монолитных, проектируются так, чтобы содержать именно то, что нужно, без каких-либо дополнительных издержек, замедляющих обработку.

Микросервисы должны быть небольшими и выполнять определенные бизнес-функции, которые часто разрабатываются разными командами, но развертываются одновременно. Поэтому небольшая команда может создавать конкретный сервис с использованием контейнеров с желаемым языком программирования, библиотеками, стилем реализации API и методами масштабирования. Они могут свободно развертывать свои бизнес-функции посредством отправки контейнера в реестр и развертывания CICD, не влияя на другие сервисы в сетке взаимодействующих сервисов. В качестве альтернативы другие небольшие команды могут разрабатывать собственные сервисы, и им не нужно беспокоиться об API других сервисов.

Применение контейнеров для микросервисов обеспечивает гибкость и высокие темпы работы для небольших групп, а также позволяет команде стандартизации архитектуры внедрять рекомендации, которым должны следовать все сервисы, не влияя на деятельность команды разработчиков. Например, структура ведения журнала, подход к безопасности секретных данных, слой оркестрации и технология CICD могут быть обязательными, но выбор языка программирования отдан на откуп команде программистов. Это позволяет централизованно управлять составной системой и разрабатывать каждый сервис индивидуально.

Гибридное и миграционное развертывания приложений

С помощью контейнеров запуск гибридных архитектур или поддержка миграции рабочих нагрузок в облако осуществляются быстрее и проще, чем с помощью большинства других методов. Поскольку контейнер представляет собой отдельную единицу, то независимо от того, развернут он локально (например, в частном облаке) или в общедоступном облаке (например, AWS), он будет одним и тем же. Кроме того, данный подход позволяет группе разработчиков архитектуры развернуть полностью составное приложение (несколько микросервисов, контейнерные приложения и т. д.) в локальной среде и использовать облачную среду в качестве площадки *аварийного восстановления (disaster recovery, DR)* или переключения и наоборот. Он также позволяет переносить контейнеризованное приложение из локальной среды в облако, почти не рискуя, что делает его предпочтительным способом выполнения крупномасштабных миграций после контейнеризации приложений. К сожалению, не все так просто. Предусмотрены изменения, необходимые для поддержки компонентов, составляющих систему, в зависимости от того, какое облачное решение и решение оркестрации применяются. Например, при использовании Kubernetes как локально, так и в облаке могут потребоваться изменения в файле манифеста Kubernetes для поддержки различных методов ввода, томов хранения, балансировщиков нагрузки и других компонентов.

Возможны и более сложные варианты, в том числе выполнение активных рабочих нагрузок в локальной среде и у выбранного поставщика облачных услуг или перемещение контейнеров из различных облаков (часто называемых мультиоблаками). Целями реализации такой архитектуры могли быть использование преимуществ ценообразования или уход от проблемных облаков (например, с задержками, неработающими центрами обработки данных и т. д.), чтобы при этом обеспечивалась непрерывность бизнес-процессов. Однако сложность такого типа распределения рабочей нагрузки по облакам очень высока и будет невозможна для многих приложений. Только рабочие нагрузки, специально спроектированные для обеспечения надлежащей обработки сохраняемости данных и *оперативной обработки транзакций (online transaction processing, OLTP)*, будут реализованы без ошибок. На этом этапе развития инструментов облачных брокеров, распределенных баз данных и задержек в сети рекомендуется использовать только одну облачную платформу для приложений cloud native.

Недостатки контейнеров

В мире облачных вычислений контейнеры не только являются популярным способом создания архитектур cloud native, но и быстро становятся стандартом для проектирования микросервисов. Тем не менее их стоит применять не во всех случаях, в частности, не нужно задействовать один контейнер для нескольких задач. Понятие задач, или областей фокусировки компонента, заключается в том, что каждый модуль или класс должен отвечать за единственную область функциональности.

Если говорить о контейнерах, множественные задачи означают, что веб-сервер и база данных работают в одном контейнере. Если придерживаться сервисного подхода, разбиение монолитного сервиса на отдельные компоненты обеспечит меньший масштаб потенциальных проблем (blast radius) приложений и их будет проще развертывать и масштабировать независимо. Это не означает, что у контейнера должен быть только один поток, но у него должна быть лишь одна роль. Контейнеры могут совместно использовать хранилища данных и взаимодействовать между собой, но размещать несколько компонентов или задач в одном контейнере неправильно. Проще говоря, контейнеры — это не виртуальные машины.

Отношение к контейнерам как к виртуальным машинам не ограничивается развертыванием нескольких задач в одном контейнере. Разрешение демону SSH (или другим консольным приложениям) иметь доступ к контейнеру отрицательно сказывается в первую очередь на назначении контейнера. Поскольку контейнеры — дискретные компоненты, они должны быть настроены и сконфигурированы, а их код следует развернуть как часть начальной фазы разработки. Он, в свою очередь, становится артефактом развертывания, который передается в хранилище и развертывается с помощью используемого инструмента оркестрации. Разрешение прямого консольного доступа к контейнеру означает, что в него могут быть внесены изменения, контейнер больше не является неизменяемым компонентом и нельзя быть уверенными в согласованности содержимого. Контейнер нуждается в обновлении в связи с исправлением ошибок или внесением улучшений, и это обновление лучше выполнить в базовом контейнере, протестировать по соответствующим каналам и отправить в репозиторий для развертывания.

Если приложению требуется выполнение нескольких задач или доступ по SSH, рассмотрите возможность использования вместо контейнеров облачных экземпляров виртуальных машин. В качестве альтернативы, чтобы точнее следовать принципу cloud native, поработайте над приложением: разделите его или усовершенствуйте конвейер CICD и тем самым обеспечьте более быстрые и согласованные процессы развертывания.

Бессерверность

Термин «бессерверность» не означает отсутствие серверов — он означает, что ресурсы могут использоваться без оглядки на производительность сервера. Бессерверные приложения не требуют, чтобы команда, занимающаяся проектированием или эксплуатацией, предоставляла, масштабировала или обслуживала серверы. Все эти работы выполняет облачный провайдер. При таком подходе, вероятно, впервые в истории разработчики могут сосредоточиться именно на том, что они лучше всего умеют, — на написании кода. Огромным преимуществом бессерверных приложений по сравнению с более традиционными подходами, даже контейнерами, является не только гибкость и целенаправленность, но и снижение затрат. Ценообразование при бессерверном подходе, как правило, определяется временем выполнения

сервиса с тарификацией интервалов 100 мс. Сервисы, выполняющие фрагменты кода с такими короткими интервалами, по определению являются микросервисами и позволяют по максимуму использовать преимущества бессерверной технологии.

В большинстве случаев, когда обсуждается бессерверность, в первую очередь рассматривается шаблон «функция как услуга». У каждого из трех крупных облачных провайдеров есть свои версии: AWS Lambda, Azure Functions и Google Cloud Functions. Они ведут себя примерно одинаково и позволяют развертывать код и выполнять его как событие. Чаще всего при таком типе бессерверных сервисов место, где выполняется код, считается ключевым элементом всех бессерверных шаблонов проектирования. Однако существуют и другие облачные сервисы, которые попадают в бессерверную категорию, в большинстве случаев эти примитивы используются совместно для формирования подсистем в облачной среде. Согласно классификации AWS есть восемь областей высокого уровня, которые имеют бессерверные сервисы. Они включают в себя вычисления, прокси API, накопители, хранилища данных, обмен сообщениями между процессами, оркестрацию, аналитику и инструменты для разработчиков. Задействуя один или несколько сервисов в каждой из этих категорий, можно создать сложные системы, которые для достижения того же бизнес-результата окажутся более дешевыми, более масштабируемыми и будут иметь значительно меньшие накладные расходы на управление, чем традиционные сервисы.

Масштабирование

Бессерверные приложения способны масштабироваться автоматически, не требуя от вас понимания того, насколько производительны отдельные серверы. Важным фактором для этих сервисов являются единицы потребления — обычно пропускная способность или объем памяти, которые могут автоматически регулироваться в зависимости от потребностей приложения. Кроме того, кратковременные функции, такие как AWS Lambda, работающие не более пяти минут, не масштабируются автоматически, как приложение, развернутое на инстансе облачного сервера при интенсивной загрузке ЦП. Функции должны вызываться в ответ на события; они выполняются ровно один раз и затем прекращают свою работу. Следовательно, возможности масштабирования функций зависят от природы запускающих их событий, а не от добавления дополнительных инстансов через группу автоматического масштабирования. Функции предназначены для частого запуска источником события, например каждый раз, когда сообщение помещается в очередь или поток, что может происходить тысячи раз в секунду.

Например, возьмем типичный кластер веб-серверов, запускающий Apache для обслуживания главной страницы веб-приложения. Каждый раз, когда пользователь отправляет запрос на сайт, создается рабочий поток, который обслуживает его и ожидает дополнительных взаимодействий. По мере увеличения числа пользователей создается все больше потоков, в итоге инстанс работает на полную мощность и начинают подключаться дополнительные инстансы, создаваемые

с помощью группы автоматического масштабирования, обеспечивая больше потоков для обслуживания пользователей. Этот подход работает отлично, однако он не предусматривает наращивание ресурсов; вместо этого при необходимости выделяется целый сервер, за работу которого компании приходится платить поминутно. Та же самая конструкция с применением прокси-сервера API и функции будет по-прежнему в состоянии динамически обслуживать пользователя, но поток или функция выполняются только при вызове и затем исчезают, а компания платит за каждые 100 мс их работы.

Шаблоны бессерверного подхода

Способы применения бессерверных технологий ограничены только воображением команды разработчиков, вовлеченных в создание системы. Каждый день разрабатываются новые методы, которые еще больше расширяют возможности того, чего можно достичь с помощью бессерверных приложений, и благодаря быстрым темпам инноваций, внедряемых поставщиками облачных услуг, количество новых функций и услуг в этой области постоянно растет. Тем не менее данной технологии уже несколько лет, и за это время выработалось несколько основных походов к проектированию бессерверных решений, а именно обработка веб- и бэкенд-приложений, обработка данных и пакетов, а также автоматизация системы.

Обработка веб- и бэкенд-приложений

Самый простой способ описать этот шаблон проектирования — сравнить его с традиционной трехуровневой архитектурой приложения. В течение многих лет доминирующий подход к масштабируемому веб-приложению заключался в настройке балансировщика нагрузки, веб-серверов, серверов приложений и баз данных. Существует множество типов сторонних инструментов и методов для поддержки данной конструкции, и если все сделано правильно, можно создать отказоустойчивую архитектуру, которая будет масштабироваться, чтобы выдержать скачок трафика в приложении. Сложность такого подхода заключается в том, что он не позволяет в полной мере пользоваться преимуществами облачных сервисов, вынуждая нас полагаться на планирование емкости, количество специалистов для управления средой и множество настроек для обеспечения правильной конфигурации кластера и отработки отказа. Точно такое же трехуровневое приложение может быть создано с помощью бессерверных технологий, которые делегируют управление, выполнение приложений, обеспечение отказоустойчивости и другие административные функции сервису и при этом обходятся намного дешевле.

Мы имеем полностью бессерверную реализацию трехуровневого приложения. Статичная веб-страница размещается в хранилище объектов Amazon S3, где предоставляется пользователям, которые вызывают ее с помощью своих браузеров. Поскольку этот сервис позволяет обслуживать страницы, он автоматически обрабатывает все требования к распределению нагрузки и вычислительной мощ-

ности и не нуждается в настройке веб-сервера для размещения веб-приложения. Когда взаимодействие пользователя со страницей необходимо обрабатывать динамически, сервис делает API-запрос к Amazon API Gateway, который действует как масштабируемая точка входа для кода, динамически выполняющего пользовательский запрос с помощью функции AWS Lambda, вызываемой событием API-вызова. Функция Lambda обладает практически неограниченными возможностями выполнять другие действия. В данном случае она обновит таблицу Amazon DynamoDB для сохранения данных. Другие функциональные возможности бэкенда могут быть реализованы с помощью разветвленных функций, таких как помещение сообщений в очередь, обновление темы для дальнейшего обмена сообщениями или запуск совершенно новой внутренней подсистемы для выполнения любых необходимых действий.

Данные и пакетная обработка

Одно из самых больших преимуществ использования облака — это возможность быстро и эффективно обрабатывать большие объемы данных. Двумя наиболее распространенными высокоуровневыми подходами являются обработка данных в режиме реального времени и пакетная обработка. Как и в предыдущем примере, все эти методы существуют на протяжении длительного времени, и для их реализации можно использовать более традиционные инструменты и подходы. Однако внедрение бессерверных технологий позволяет ускорить бизнес-процессы и снизить денежные расходы.

Например, рассмотрим обработку и анализ потока данных из социальных сетей в режиме реального времени, это может быть также анализ потока щелчков кнопкой мыши на всех веб-ресурсах для поддержки контекстной рекламы. Поток щелчков кнопкой мыши или других данных поступает через сервис потоковой передачи данных Amazon Kinesis, выполняющий функцию Lambda для каждого микропакета данных, которые обрабатываются и анализируются в зависимости от требований. Затем функция Lambda сохраняет в таблице DynamoDB важную информацию, такую как результаты анализа тональности текста (если задействуется поток данных из социальных сетей) или статистика щелчков по популярным рекламным объявлениям (если применяется поток щелчков). После сохранения метаданных в таблицу можно спроектировать и реализовать еще одну подсистему с дополнительными отчетами, анализом и панелями мониторинга, используя новые бессерверные сервисы.

Чтобы показать, как происходит пакетная обработка данных, рассмотрим случай, когда пакетные файлы помещаются в хранилище объектов Amazon S3. Это могут быть какие угодно файлы: изображения, файлы CSV или другие большие двоичные объекты, которые необходимо каким-либо образом обработать или изменить до завершения обработки данных. Как только файл помещен в хранилище объектов, функция AWS Lambda выполняет и изменяет его в соответствии с нужными

задачами. Пакетный процесс может включать в себя выполнение других подсистем, которые используют данные, модификацию изображения для изменения его размера или частичного обновления различных хранилищ данных. В этом примере таблица DynamoDB обновляется, сообщение помещается в очередь, а пользователю отправляется уведомление.

Система автоматизации

Еще один популярный бессерверный шаблон проектирования, предназначенный для выполнения регулярных процедур по обслуживанию и эксплуатации среды. Функции соответствуют требованиям модели CNMM относительно полной автоматизации, поэтому отлично подходят для выполнения задач, предусмотренных системой. По мере роста облачных ландшафтов автоматизация обеспечит согласованность критически важных задач и не потребует от организации увеличения команды эксплуатации, поддерживающей среду. Возможности реализации сервисов, выполняющих задачи администрирования, безграничны, они ограничены только требованиями и творческим мышлением команды эксплуатации.

Некоторые распространенные случаи применения администрирования — это настройка функций, которые выполняются по расписанию, и проверка каждого инстанса во всех учетных записях на предмет правильной маркировки, правильного размера ресурсов инстанса, остановки неиспользуемых инстансов и очистки незадействованных устройств хранения. Бессерверные приложения могут использоваться не только для выполнения рутинных задач администрирования, но и для обеспечения безопасности. В качестве примеров можно привести использование функций для выполнения и проверки шифрования объектов, помещенных в хранилище или на диски, проверку правильности применения политик учетных записей и их сопоставление с нестандартными политиками соответствия на серверах или других системных компонентах.

Бессерверные системы и задачи

Бессерверность — это все больше распространяющийся важный способ достижения успеха в работе с архитектурой cloud native. Инновации в этой сфере происходят постоянно, и так будет продолжаться в течение многих лет. Тем не менее есть методы, которые не применяются непосредственно к бессерверным архитектурам. Например, длительные запросы зачастую оказываются неудачными, учитывая недолгое время жизни функции (менее 5 мин). В тех случаях, когда запрос занимает больше времени, чем используется сервис функций, возможной альтернативой становятся контейнеры, поскольку у них нет времени жизни, как у функций.

Другим примером задач для бессерверных систем является случай, когда источник события выполняется не так, как ожидалось, следовательно, функция запускается чаще, чем предполагалось. В традиционных серверных приложениях этот сценарий

будет перегружать ЦП и приведет к зависанию сервера и остановке. Поскольку каждая функция является дискретной единицей, в бессерверной среде они никогда не прекратят выполнение, даже если желательна остановка. В лучшем случае это приведет к увеличению стоимости этой функции, в худшем — к повреждению или потере данных в зависимости от характера функции. Для предотвращения этого следует провести тестирование и убедиться, что триггеры подходят для нее. Кроме того, для обеспечения отказоустойчивости устройства рекомендуется настроить оповещения об общих вызовах для этой функции, которые уведомят, если она выйдет из-под контроля, и позволят персоналу вмешаться и остановить обработку.

Наконец, важно знать сервисы поставщика облачных вычислений и разбираться, какие из них работают совместно друг с другом. Например, одни сервисы AWS являются источниками триггеров для Lambda, другие — нет. Amazon Simple Notification Service может запускать функцию Lambda каждый раз, когда в тему помещается сообщение, а сервис Amazon Simple Queue не может. Следовательно, знание того, как сервисы сочетаются друг с другом, и возможность выбора между вызовом функции или принятием собственного решения гарантируют, что подсистемы будут сочетаться друг с другом и работать с использованием наилучших практик.

Платформы и подходы к разработке

Существует множество платформ и подходов, используемых для разработки горизонтальных масштабируемых приложений в облаке. Из-за особенностей облачных провайдеров у каждого есть специфические сервисы и платформы, которые работают лучше всего. В главе 9 подробно описано, как их следует выбирать, на примере AWS. В главе 10 рассматривается подход Microsoft Azure. Глава 11 разбирает шаблоны проектирования и платформы, которые лучше всего подходят для этого облака. Несмотря на свою схожесть, все они имеют определенные особенности и будут по-разному реализовываться каждой командой разработчиков.

Резюме

В данной главе мы изучили разработку архитектур cloud native с использованием микросервисов и бессерверных вычислений в качестве принципа проектирования. Смогли понять, в чем разница между SOA и микросервисами, что такое бессерверные вычисления и как они вписываются в архитектуру cloud native. Мы узнали, что у каждого из поставщиков облачных решений есть уникальный набор платформ и подходов, которые можно применять для разработки масштабируемых архитектур. В следующей главе вы узнаете, как выбрать эффективный технологический стек.

4 Как выбрать технологический стек

Мир облачных вычислений огромен, и хотя среди поставщиков облачных технологий есть несколько доминирующих игроков, у этой экосистемы есть другие области, которые имеют решающее значение для успеха. Как решить, какого поставщика облачных услуг выбрать? Каких партнеров следует рассматривать и какие решения они готовы предложить? Изменится ли система закупок в облаке или останется прежней? До какой степени вы готовы довериться поставщику облачных технологий в вопросах управления сервисами? Все это важные и правомерные вопросы, на которые в этой главе будут даны ответы.

Экосистемы облачных технологий

Изучение возможностей облачной экосистемы и способов их применения — важнейший шаг при переходе в облачную среду. Есть три основных направления, на которых следует сфокусироваться, выбирая партнеров в этом путешествии: провайдеры облачных решений, независимые поставщики программного обеспечения и системные интеграторы. Вместе со штатными сотрудниками, участвующими в этом переходе, они составят основу кадрового, процессуального и технологического потенциала компании для ее преобразования с помощью облачных вычислений.

Общедоступные облачные провайдеры

Данная книга посвящена общедоступным облачным провайдерам, которые, скорее всего, будут доминировать на рынке ИТ-услуг. Облако, каким оно существует сегодня, появилось в 2006 году, когда платформа *Amazon Web Services (AWS)* запустила свои первые общедоступные сервисы (Amazon Simple Queuing Service и Amazon Simple Storage Service). После этого она спешно начала внедрять инновационные решения с инстансами виртуальных серверов, виртуальными сетями, блочным хранилищем и другими базовыми инфраструктурными сервисами. В 2010 году компания Microsoft выпустила платформу Azure, функции которой подобны тем, что предлагали в AWS, и начала конкурентную борьбу за клиентов в этой сфере. В это же время компания Google стала широко предлагать свои услуги на основе

платформы, которая в итоге переросла в Google Cloud Platform (GCP) (облачная платформа Google), существующую и поныне.

Несмотря на то что существуют и другие облачные решения, ориентированные на разные узкоспециализированные области и подходы к удовлетворению потребностей клиентов, три перечисленных доминируют на мировом рынке облачных услуг. Принятие решения о поставщике облачных услуг при первоначальном рассмотрении предложений может показаться сложным главным образом потому, что более мелкие игроки пытаются позиционировать себя так, будто с ними выгоднее иметь дело, так как они публикуют информацию о доходах или доле рынка, относящиеся к услугам, которые в действительности не имеют никакого отношения к облаку. Поскольку не существует единого определения облака, а также методики подсчета доходов, связанных с ним, каждая компания может как угодно рекламировать свои сервисы. Поэтому, прежде чем принимать решение, важно понять масштаб поставщика облачных услуг. Каждый из трех основных поставщиков облачных услуг предоставляет очень привлекательные базовые сервисы, часто называемые «инфраструктура как сервис», и дополняет их очень полезными управляемыми облачными предложениями, иногда называемыми «платформа как сервис» (PaS), которые содержат все, от баз данных до сервисов приложений, инструментов DevOps и сервисов искусственного интеллекта и машинного обучения.

Существует множество критериев, которые необходимо учитывать при выборе поставщика облачных вычислений. Часть из них связаны с технологиями, которыми клиенты пользовались в прошлом и с которыми им удобно работать. Рассмотрим основные характеристики.

- *Масштаб*. Бизнес в сфере облачных вычислений — это масштабирование и возможность доставки в любое время и в любое место. Даже если клиенту в настоящее время не нужны облачные ресурсы в любой точке мира или для массового предоставления в одном географическом регионе, выбор поставщика облачных вычислений, который может это сделать, крайне важен: правильный выбор позволит получить опыт и сэкономить деньги и появится желание продолжать расти и внедрять инновации.
- *Безопасность/соответствие требованиям*. Приоритетом для всех поставщиков облачных вычислений должна быть безопасность. Если провайдер облачных услуг, получив большинство сертификатов соответствия, перестанет уделять этому должное внимание, лучше выбрать другого поставщика, который делает это направление приоритетным.
- *Богатство возможностей*. Темп разработки инноваций постоянно растет, и поставщики облачных услуг обращают внимание на области, которые даже не рассматривались несколько лет назад. Будь то машинное обучение, блокчейн, бессерверность или какая-либо другая новая технология, необходимо выбирать тех поставщиков облачных технологий, которые постоянно внедряют инновации.

- *Стоимость*. Хотя это и не единственная причина выбора поставщика, стоимость предлагаемых услуг всегда нужно учитывать. Вопреки распространенному мнению, рынок поставщиков облачных услуг не является ареной соревнования за минимальную предлагаемую цену. Масштаб и инновации дают возможность поставщикам снижать цены, позволяя клиентам экономить (еще одна причина, по которой масштаб имеет значение). На данный момент все три крупнейших поставщика облачных вычислений конкурентоспособны по стоимости, но важно постоянно следить за ценами.

В течение многих лет компания Gartner глубоко анализировала рынок поставщиков облачных услуг. Последнюю версию магического квадранта Gartner для сервисов облачной инфраструктуры можно найти здесь: https://www.gartner.com/en/documents/3989743.

Независимые поставщики программного обеспечения и технологические партнеры

Возможности, предоставляемые провайдерами облачных технологий, — это база для любой стратегии миграции в облако, но основные компоненты все-таки формируют независимые разработчики ПО (ISV) и технологические партнеры. Во многих случаях поставщики облачных решений смогут предоставить встроенные инструменты, которые удовлетворят все потребности клиентов. Однако возникают ситуации, когда эти инструменты не способны закрыть все существующие вопросы, и именно в таких случаях на помощь приходят независимые поставщики программного обеспечения и технологические партнеры. Даже если у поставщика облачных услуг есть специальный сервис для решения задачи, по различным причинам клиенты зачастую предпочитают использовать сторонние инструменты. Во-первых, такие инструменты часто оказываются более продвинутыми или многофункциональными, поскольку они разработаны специально для облака и широко применяются. Во-вторых, компания может иметь обширный опыт работы с определенными продуктами ISV, что позволяет облегчить принятие решения о переходе в облако. И наконец, во многих случаях поставщики облачных услуг не имеют сопоставимых сервисов, поэтому обращение к сторонним поставщикам позволит компаниям решать проблемы, не вынуждая их вести с нуля собственные разработки.

На этом этапе развития облачных вычислений подавляющее большинство традиционных независимых поставщиков программного обеспечения решили принять облачную стратегию. Подобно другим компаниям, они потратили много времени и усилий на оценку поставщиков облачных услуг и остановились на каком-то одном из них (или нескольких, в зависимости от типа продукта). Стандартные модели использования продуктов ISV включают развертывание продукта, управляемого заказчиком или работающего по принципу «программное обеспечение как сервис» (Software as a Service). Как правило, независимый поставщик программного обеспе-

чения или технологический партнер уже определились с моделью ценообразования, которая напрямую связана с моделью потребления, предлагаемой покупателю.

Продукты, управляемые клиентами

Если развертываются продукты, управляемые клиентами, независимый поставщик программного обеспечения решает либо создать продукт заново, либо перенести свой уже существующий в облачную модель. Зачастую это означает, что поставщик предпринял серьезные шаги, чтобы самостоятельно адаптировать свой продукт к архитектуре cloud native и затем продать его конечному потребителю. В простейшей форме продукт тестируют один или несколько поставщиков облачных услуг, чтобы убедиться, что он работает, как задумано, и клиент может установить и настроить его так же, как это делается с локальной версией. Примерами такого подхода могут быть SAP или различные продукты Oracle. В более сложных случаях независимый поставщик программного обеспечения изменяет архитектуру продукта так, чтобы использовать преимущества сервисов и микросервисов поставщика облака для достижения такого уровня масштаба, безопасности или доступности, который было бы трудно получить, используя тот же продукт офлайн. Независимо от подхода, который избрал независимый поставщик программного обеспечения, выбор правильного продукта требует тщательной оценки функций и моделей ценообразования, а также сравнения с аналогичными сервисами поставщиков облачных вычислений, чтобы гарантировать, что продукт масштабируем и безопасен в такой степени, которая необходима для архитектуры cloud native.

Программное обеспечение как сервис

Независимые поставщики программного обеспечения и технологические партнеры часто решают полностью изменить дизайн своих предложений, ориентированных на предоставление услуг, так, чтобы клиенту было максимально легко использовать их. Чтобы сделать это, поставщики почти всегда выбирают конкретного поставщика облачных вычислений и поверх его предложения разрабатывают собственное, делая его доступным для клиентов, но не предоставляя доступ к базовой инфраструктуре или к сервисам поставщика облачных вычислений. Очень крупные поставщики, учитывая их компетенцию в управлении гипермасштабным облаком, фактически предпочитают запускать собственное частное облако или локальные среды, которые интегрируются с облачными сервисами, выбранными заказчиками.

Программное обеспечение как сервис (SaaS), которое в значительной степени представляет собой облачные вычисления, не является основным продуктом крупных поставщиков облачных услуг и не должно учитываться при оценке такого поставщика для большинства проектируемых рабочих нагрузок. Некоторые крупные поставщики тоже размещают в своих облаках приложения SaaS (например, Office 365 или LinkedIn в случае с Microsoft), аналогично тому, как это делают независимые поставщики программного обеспечения, и при выполнении интеграции их

услуги тоже можно рассматривать. Тем не менее поставщики SaaS для многих клиентов играют важнейшую роль при переходе к cloud native, поскольку предлагают свои сервисы непосредственно клиентам независимо от выбранного ими облака. Интеграция предложений SaaS с рабочими нагрузками клиентов и хранилищами данных — один из наиболее распространенных способов использования этих сервисов в сочетании со специально разработанными облачными рабочими нагрузками. Примерами крупных предложений SaaS являются Salesforce и WorkDay, которые предлагают очень сложную бизнес-логику, хорошо сочетающуюся с системами компаний.

Во многих случаях у независимых поставщиков ПО или технологических партнеров имеются предложения, среди которых заказчик может выбрать версию с самостоятельным управлением и развернуть ее в конкретном облаке или использовать версию SaaS того же предложения. Существует множество причин, по которым продавец может решить сделать свои предложения доступными таким образом. Часто клиенты желают сохранить данные и контроль над продуктом в своей среде либо уже имеют лицензии, которые данный продукт использует. В других случаях заказчик хочет быстро перейти к работе, и, поскольку у него еще нет программного обеспечения, он обращается непосредственно к версии SaaS, поэтому ему нужно лишь интегрировать продукт со своими системами. В конечном счете независимые поставщики программного обеспечения и технологические партнеры должны удовлетворять запросы своих клиентов, и это в первую очередь следует учитывать при выборе поставщика.

Консалтинговые партнеры

Консалтинговые партнеры, или *системные интеграторы (system integrators, SI)*, существуют практически с момента зарождения информационных технологий. Дело в том, что у компаний зачастую нет времени, ресурсов или желания осваивать новые технологии или увеличивать количество рабочей силы, чтобы удовлетворить запросы своего бизнеса. Поэтому SI-партнеры заполняют такие пробелы и помогают компании двигаться вперед, предоставляя специалистов, способных показывать быстрые результаты, доводить проекты до конца и достигать поставленных бизнес-целей. Это партнерство может принимать различные формы и размеры и быть разовым или долгосрочным и стратегическим в зависимости от того, как отдельная компания решает вести свой бизнес. Как правило, SI-партнеры могут рассматриваться как нишевые, региональные или глобальные игроки, каждый из которых занимает ключевое место при переходе компании в облако.

Нишевые SI-партнеры

Нишевые SI-партнеры — это компании, которые предлагают специфические сервисы для определенного типа технологии, области деятельности или облака. Ча-

сто это относительно небольшие по размеру компании, но в них обычно трудятся наиболее опытные специалисты в своих областях. Они работают как предметные эксперты при реализации крупных проектов или конкретного проекта в своей области. На нынешнем этапе развития облачных вычислений существуют нишевые SI-партнеры, которые специализируются на конкретных поставщиках облачных услуг и даже на конкретных сервисах у одного или всех поставщиков облачных вычислений. В первую очередь при выборе такого партнера следует обратить внимание не на его масштаб, а на степень профессионализма в выбранной области.

При оценке нишевых SI-партнеров клиенты должны учитывать сертификацию отдельных ресурсов по соответствующим технологиям, сроки поставки и отзывы клиентов, для которых была проделана аналогичная работа. Нишевых SI-партнеров часто подключают для получения консультаций по проектированию архитектур cloud native, определения требований к архитектуре больших данных и структуре данных, а также обсуждения вопросов безопасности и соответствия для конкретных отраслей или требований.

Иногда SI-партнеры, с которыми вы сотрудничаете, могут поручать своим сотрудникам задачи, выходящие за рамки их основной специализации, и это плохой признак.

Региональные SI-партнеры

Региональные SI-партнеры пользуются популярностью у компаний, работающих в определенных географической области или регионе, поскольку у такого партнера все ресурсы обычно сосредоточены там. Часто рассматриваемые как стратегические партнеры с взаимоотношениями на уровне руководителей высшего звена, эти партнеры помогают клиенту определить свою стратегию и разработать конкретные проекты с помощью собственных ресурсов. Как правило, в таких компаниях-партнерах трудятся грамотные технические специалисты, и масштаб этих компаний достаточен для того, чтобы иметь возможность решать для заказчика крупные и сложные задачи, включая управление проектами, создание технической архитектуры, разработку и тестирование. Как и любые партнеры, региональные SI-партнеры часто специализируются на конкретных вертикалях, технологиях/облаках или других областях, которые позволяют им выделиться на фоне конкурентов. Они также очень чутко реагируют на потребности клиентов и могут стать определяющим фактором успешного или провального перехода в облако. Пример регионального SI-партнера, помогающего клиентам перейти к облаку, — Slalom.

Недостатком сотрудничества с региональными SI-партнерами является то, что некоторые из них привлекаются к поддержке глобальных инициатив, а те могут потребовать участия тысяч людей или наличия сверхдолгосрочных планов развития компании.

Глобальные SI-партнеры

Глобальные SI-партнеры популярны благодаря своим масштабам и способности поддерживать самые крупные, длительные и сложные проекты клиентов. Как следует из названия, они носят глобальный характер, действуют в большинстве стран мира и имеют значительное присутствие во всех регионах. С самого начала глобальные SI-партнеры играли роль в делегировании внешним подрядчикам обязанностей ИТ-отдела, а также проектирования, стратегического планирования и доставки продуктов, позволяя как крупным, так и малым компаниям сосредоточиться на своем основном бизнесе.

Глобальные SI-партнеры часто сотрудничают с компанией на уровне совета директоров и участвуют в формировании ее долгосрочной стратегии развития на протяжении десятилетий, предоставляя ресурсы и технологии, соответствующие требованиям заказчика. Они выполняют сложные глобальные развертывания для нескольких облачных провайдеров, географических регионов или технологий. Примером глобального SI-партнера, помогающего клиентам в переходе к cloud native, является Accenture.

Недостаток сотрудничества с глобальным SI-партнером заключается в том, что качество иногда может быть принесено в жертву масштабу, так как для обеспечения глобального охвата задействуется большое количество ресурсов более низкого уровня. Кроме того, из-за своего глобального характера такие партнеры часто реализуют важные проекты в регионе, где клиент не ведет деятельность, что усложняет координацию.

Закупки в облаке

Одна из причин популярности облака и того, что архитектуры cloud native становятся обычным методом проектирования рабочих нагрузок, связана с используемой моделью закупок и потребления. Если говорить подробнее, это возможность относить закупки к *операционным расходам (operational expenditures, OpEx)* вместо *капитальных (capital expenditures, CapEx)*, что может оказать большое влияние на то, как компания организует доходы, налоги и долгосрочную амортизацию активов. Однако это всего лишь поверхностное воздействие облака на то, как организации приобретают свои ИТ-ресурсы.

Переход поставщиков облачных услуг на схему оплаты по факту использования позволил независимым поставщикам программного обеспечения аналогичным образом перестроить свою бизнес-модель, что упростило использование их продуктов, как и выбранных сервисов облачных поставщиков. Это изменение может серьезно повлиять на заключение организациями контрактов с независимыми

поставщиками программного обеспечения (при этом стоимость услуг часто снижается из-за того, что оплачиваются только использованные ресурсы), ведь теперь становятся невыгодны долгосрочные контракты, учитывающие количество задействованного оборудования. Как и сторонние независимые поставщики ПО, которые изменяют свою модель закупок, поставщики облачных услуг часто предоставляют облачную платформу, в рамках которой независимые поставщики могут составлять цифровые каталоги своих продуктов, что упрощает клиентам поиск, тестирование, покупку и развертывание программного обеспечения.

Все эти изменения предназначены для того, чтобы клиенты могли в корне изменить свое покупательское поведение на более комфортное, с быстрой подстройкой под бизнес-требования и простым определением предложений, которые помогут решить их бизнес-проблемы.

Облачные рынки

Использование рынка поставщиков облачных услуг в качестве места для поиска и приобретения программного обеспечения, необходимого для работы в облаке, — это один из способов получения компаниями нужных инструментов в любой момент. По сути, клиенты стремятся либо разработать новые рабочие нагрузки, либо перенести существующие в облако, модернизировать рабочую нагрузку, если она была перенесена, а затем управлять масштабом своей среды за счет хорошо организованной рабочей силы. Чем крупнее компания, тем более сложными могут стать ее бизнес-требования из-за широкого круга заинтересованных сторон, бизнес-единиц и категорий потребителей. На то, чтобы при покупке ISV-продуктов как следует подготовить условия предоставления услуг, требования к масштабу, модель тарификации и т. д., могут уйти месяцы работы. Торговые площадки поставщиков облачных услуг — один из инструментов, который помогает компаниям свести к минимуму эти препятствия на пути к использованию необходимого им программного обеспечения.

В зависимости от уровня торговых площадок компания может выбирать из тысяч различных пакетов программного обеспечения практически в каждой категории. В результате одного нажатия кнопки или вызова API можно купить и развернуть ISV-продукт в облачной отправной точке клиента, причем сделать это за считаные минуты, протестировав и при необходимости интегрировав ПО в среду. Закупка ПО на облачном рынке имеет дополнительное преимущество, заключающееся в том, что программное обеспечение сразу настроено и соответствует спецификациям поставщика. Как правило, для его работы требуется незначительная дополнительная настройка или не требуется вовсе. Облачные сервисы по-прежнему оплачиваются отдельно (например, инстансы, хранилище, использование сети и т. п.), и в большинстве случаев программное обеспечение представляет собой просто дополнительную строку в счете поставщика облачных услуг.

Рынки и каталоги сервисов

Облачные поставщики часто имеют каталог сервисов, который позволяет хранить и развертывать предварительно созданные стеки приложений или сложные решения, когда это требуется пользователям с соответствующими разрешениями. Этот подход в сочетании с рынком предлагает мощный способ, позволяющий командам разработчиков сервисов выбрать правильное программное обеспечение, разместить его в каталоге и либо сразу развернуть его, либо заставить команду эксплуатации сделать это в соответствии с рекомендациями компании. На рис. 4.1 показано, в каких ситуациях рынок и предложение каталога сервисов могут вписываться в стратегию закупок организации.

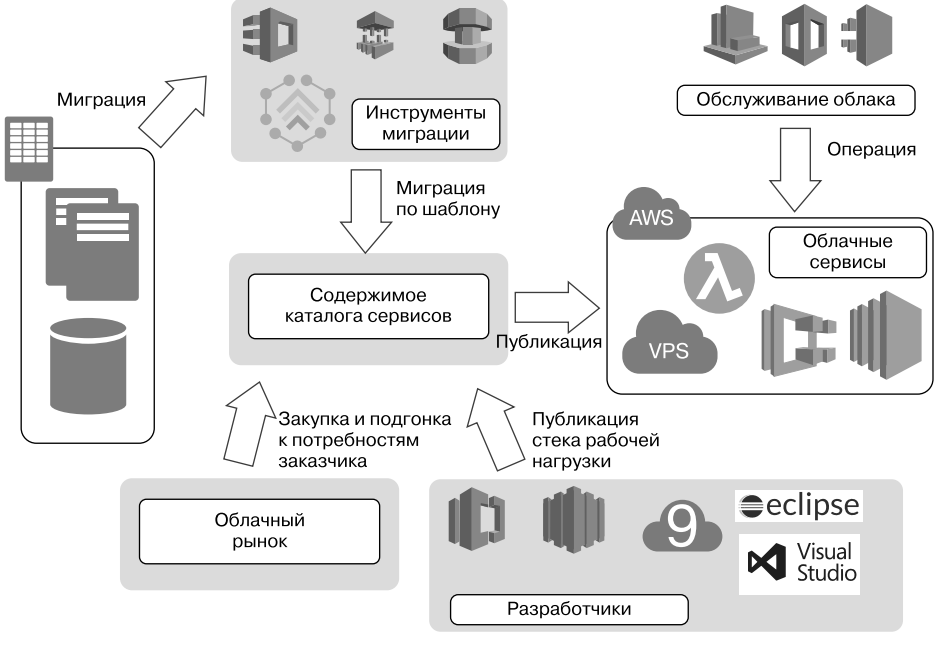


Рис. 4.1

В данном примере каталог сервисов выступает в качестве промежуточной локации компании, которая публикует отчеты об изменении поведения клиентов, приобретает программное обеспечение облачного рынка или переносит локальные рабочие нагрузки перед отправкой в целевую облачную среду.

Недостатки облачного рынка

Облачный рынок может стать мощным инструментом, позволяющим компаниям приобретать ПО практически без долгосрочных переговоров. Он позволяет компаниям тестировать новое программное обеспечение для определения того, на-

сколько легко оно решит бизнес-проблему и не потребует ли существенных первоначальных затрат. Однако бывают случаи, когда использование торговой площадки неуместно. Например, если компания уже имеет долгосрочные стратегические отношения с поставщиком ПО, который предоставляет ей индивидуальные патчи или другие специфические продукты, то обычно предпочтительнее развертывание локального программного обеспечения. В качестве альтернативы, если у компании особенно сложный вариант использования (крупные масштабы, дополнительная безопасность и т. д.) и есть ключевые знания для самостоятельного управления развертыванием, она, как правило, предпочитает отказаться от рынка и разработать и развернуть программное решение самостоятельно.

Рассмотрение лицензирования

В предыдущем подразделе были определены некоторые области, которые следует учитывать применительно к закупкам, и обсуждалась общая модель оплаты по мере потребления. Как правило, стоимость услуг поставщика облачных вычислений рассчитывается не с точки зрения лицензии, а с точки зрения потребления, однако часто она связана с дополнительными лицензионными расходами, возникающими в зависимости от того, какое ПО используется. Традиционно лицензирование программного обеспечения основывалось на количестве процессоров или ядер, на которых оно будет установлено, что давало возможность поставщику понять объем задействования программного обеспечения. Это срабатывало в то время, когда клиенты вкладывали большие капиталовложения в оборудование и точно знали, на каких серверах какое программное обеспечение будет работать. Ввиду своей гибкости облако делает эту модель устаревшей. Но как же тогда клиенты должны оплачивать лицензии на используемое программное обеспечение? Независимые поставщики ПО все еще хотят получать плату за объем использования, поэтому им необходимо найти другие способы определения ключевых показателей потребления. Далее приведены некоторые общие характеристики.

- *Пропускная способность сети, объем хранилища или другой физический компонент*. Эти параметры показывают, сколько сетевого трафика и гигабайт памяти используется. Либо же можно использовать другие показатели оборудования, не основанные на ЦП.
- *Количество хостов*. Использование в облаке отдельных серверов по-прежнему востребовано и еще долго будет пользоваться популярностью. В зависимости от программного обеспечения взимания платы за хост будет достаточно для точного учета того, сколько программного обеспечения используется. Такая плата обычно взимается за единицу времени (например, в часах) и не привязывается к количеству процессоров.
- *Процент расходов поставщика облака*. За программное обеспечение, которое применяется во всем облачном пространстве (такое как средства мониторинга или защиты конечных точек), поставщики могут взимать небольшую долю от общих расходов клиента на облачные вычисления. Это позволяет оплачивать

услуги независимого поставщика ПО с учетом гибкого масштабирования системы.

- *По транзакции*. У некоторых независимых поставщиков программного обеспечения транзакции имеют очень высокую частоту, но в то же время очень малый размер. Подсчитывая количество транзакций и взимая небольшую сумму за каждую из них, независимый поставщик программного обеспечения реализует гибкую модель оплаты использования облака.

Модели ценообразования поставщиков облачных вычислений

Клиентам важно хорошо понимать показатель ценообразования для каждого приобретаемого элемента технологии. Это особенно верно в отношении поставщиков облачных услуг, которые могут иметь сложные модели ценообразования для новых и трудноочениваемых сервисов. Поскольку многие технологические сервисы, предлагаемые поставщиками облачных услуг, новые или реализуются совершенно по-новому, механизмы ценообразования могут сильно различаться, и нужно хорошо понимать, как часто и какая часть сервиса будет использоваться, чтобы не было неожиданностей при расчете стоимости. Для базовых инфраструктурных сервисов, таких как виртуальные инстансы или хранилища больших двоичных объектов, показатель для расчета цены обычно указывается за час или за месяц и ставка растет по мере увеличения размера. Однако для управляемых облачных сервисов ценообразование может быть еще сложнее. Далее рассматриваются примеры популярных сервисов AWS, используемых в микросервисных архитектурах cloud native, — AWS Lambda и Amazon DynamoDB.

С учетом сложности и вариативности ценообразования проектной группе очень важно понимать порядок применения этих сервисов в архитектуре. При больших масштабах системы даже небольшие изменения в размере транзакций, времени выполнения функций или других аспектов могут привести к значительному увеличению счета.

Пример: ценообразование AWS Lambda

Данный пример AWS Lambda взят непосредственно со страницы ценообразования сервиса AWS, дополнительные сведения можно найти там же.

Lambda засчитывает запрос при каждом исполнении кода в ответ на вызов или оповещение о событии, при этом учитываются и тестовые вызовы с консоли. Плата взимается на основе количества запросов к функциям и их продолжительности, то есть времени, в течение которого исполняется код.

Время обработки рассчитывается от начала исполнения кода до возврата значения или прекращения работы по иной причине с округлением до ближайшего значения, кратного 100 мс. Цена зависит от объема оперативной памяти, выделенной для функции.

В таблице показаны количество бесплатных секунд и приблизительная цена за 100 мс для различных объемов выделяемой памяти.

Память, Мбайт	Уровень бесплатного использования в месяц, с	Цена за 100 мс, долларов
128	3 200 000	0,000000208
192	2 133 333	0,000000313
256	1 600 000	0,000000417
320	1 280 000	0,000000521
2816	145 455	0,000004584
2880	142 222	0,000004688
2944	139 130	0,000004793
3008	136 170	0,000004897

Пример ценообразования. Если вы выделили для своей функции 512 Мбайт памяти, выполняли ее 3 млн раз в течение одного месяца и каждое выполнение длилось 1 с, ваши расходы будут рассчитываться следующим образом.

- Ежемесячные расходы на вычисления:
  - ежемесячная стоимость вычислений составляет 0,00001667 доллара за 1 Гбайт, а уровень бесплатного использования — 400 000 Гбайт;
  - общее количество вычислений: 3 000 000 · 1 с = 3 000 000 с;
  - общий объем вычислений: 3 000 000 · 512 Мбайт/1024 = 1 500 000 Гбайт;
  - ежемесячно оплачиваемые вычисления: общий объем вычислений – уровень бесплатного использования = 1 500 000 Гбайт – 400 000 Гбайт = = 1 100 000 Гбайт;
  - ежемесячные расходы на вычисления: 1 100 000 · 0,00001667 доллара = 18,34 доллара.
- Ежемесячная плата за запросы:
  - ежемесячная цена запросов составляет 0,2 доллара за 1 млн запросов, а бесплатно обеспечивается 1 млн запросов в месяц;
  - количество ежемесячных оплачиваемых запросов: общий объем вычислений – бесплатные запросы = 3 000 000 – 1 000 000 = 2 000 000;
  - ежемесячная плата за запросы: 2 000 000 · 0,2 доллара/1 000 000 = 0,4 доллара.
- Общая сумма расходов за месяц: ежемесячные расходы на вычисления + ежемесячная плата за запросы = 18,34 + 0,40 = 18,74 доллара.

Пример: ценообразование Amazon DynamoDB

Этот пример ценообразования Amazon DynamoDB взят непосредственно со страницы ценообразования AWS, дополнительную информацию можно найти там же.

В отличие от традиционных развертываний NoSQL, в которых вам нужно заботиться о памяти, процессоре и других системных ресурсах, которые могут повлиять на производительность, для DynamoDB нужно просто указать целевой коэффициент использования и минимальную или максимальную емкость, которая требуется для вашей таблицы. DynamoDB управляет выделением ресурсов для достижения намеченных показателей использования операций чтения и записи, а затем автоматически масштабирует вашу емкость в зависимости от потребности. Если предпочитаете вручную управлять пропускной способностью таблицы, можете напрямую указать емкость чтения и записи.

В таблице приведены основные концепции ценообразования DynamoDB[1].

Вид ресурса	**Детали**	**Цена в месяц, долларов**
Предоставленная пропускная способность (запись)	Одна единица ресурса записи (WCU) обеспечивает до одной записи в секунду, что достаточно для 2,5 млн записей в месяц	От 0,47 за WCU
Предоставленная пропускная способность (чтение)	Одна единица ресурса чтения (RCU) обеспечивает до одной операции чтения в секунду, что достаточно для 5,2 млн операций чтения в месяц	От 0,09 за RCU
Индексированное хранилище данных	DynamoDB взимает почасовую ставку за 1 Гбайт дискового пространства, которое занимает ваша таблица	От 0,25 за 1 Гбайт

Пример ручной подготовки. Предположим, что вашему приложению, работающему на востоке США (Северная Вирджиния), необходимо выполнить 5 млн операций записи и 5 млн последовательных операций чтения в день в таблицу DynamoDB при хранении 8 Гбайт данных. Для простоты предположим, что рабочая нагрузка в течение дня относительно постоянна, а размер элементов таблицы не превышает 1 Кбайт.

- *Единицы записи (WCU)* — 5 млн операций записи в день равны 57,9 операции записи в секунду. Один WCU может обрабатывать одну запись в секунду, поэтому вам нужно 58 WCU. При цене 0,47 доллара за WCU в месяц 58 WCU стоят 27,26 доллара в месяц.

[1] На момент написания текста. Актуальные цены здесь: https://aws.amazon.com/ru/dynamodb/pricing/provisioned/. Там же находятся и примеры расчета месячной стоимости. — *Примеч. пер.*

- *Единицы чтения (RCU)* — 5 млн операций чтения в день равны 57,9 операции чтения в секунду. Один RCU может обрабатывать две последовательно согласованные операции чтения в секунду, поэтому вам нужно 29 RCU. При цене 0,09 доллара за RCU в месяц 29 RCU стоят 2,61 доллара в месяц.
- *Хранение данных*: ваша таблица занимает 8 Гбайт памяти. При цене 0,25 доллара за 1 Гбайт в месяц ваша таблица обойдется в 2 доллара.

Общая стоимость составляет 31,86 доллара в месяц (27,14 доллара для пропускной способности при записи, 2,71 доллара для пропускной способности при чтении и 2 доллара за хранение индексированных данных).

Открытый исходный код

Как и в традиционных локальных средах, при переходе к cloud native клиенты имеют широкий выбор программного обеспечения. Кроме того, как уже говорилось, прежде чем принимать какое-либо решение, очень важно разобраться в вопросах лицензирования. Программное обеспечение с открытым исходным кодом может быть очень эффективным в облаке, и компании часто выбирают его для заполнения важных пробелов в своем технологическом ландшафте. Это могут быть потрясающие проекты, входящие в состав Apache Foundation, или какие-то другие продукты — при принятии решений клиенты должны учитывать все варианты.

Например, очень популярным решением для обеспечения управления информацией о безопасности и управления событиями безопасности (SIEM) является система Splunk, осуществляющая широкомасштабное агрегирование журнальных записей и управление событиями. Хотя эти решения считаются лучшими в своем классе, есть несколько похожих проектов с открытым исходным кодом, в частности стек *ELK* (сокращение от *ElasticSearch*, *Logstash*, *Kibaba*). Оба достигают схожих результатов, каждый с разными моделями ценообразования, которые могут быть привлекательны для разных типов клиентов. Зачастую альтернативное ПО с открытым исходным кодом пользуется сильной поддержкой сообщества, но имеет менее отточенную модель конфигурирования, в результате чего клиенту для его настройки и обеспечения эффективной работы требуются определенные навыки в технологической сфере.

Важно также помнить, что при использовании программного обеспечения с открытым исходным кодом бесплатной может быть только софтверная часть, а за использование физических ресурсов (например, виртуальных инстансов, хранилища, сети и т. д.) по-прежнему придется платить.

Облачные сервисы

Облачные провайдеры предоставляют множество сервисов, и с ускорением внедрения инноваций все сложнее становится правильно выбрать сервис для

решения конкретных бизнес-задач. Крупные облачные провайдеры разрабатывают свои сервисы так, чтобы они были похожи на строительные блоки, которые можно использовать вместе для решения проблем, которые, возможно, никогда не рассматривались для этого конкретного сервиса. Это позволяет командам заказчика, занимающимся проектированием, проявлять творческий подход и нестандартно мыслить, в чем им помогают эксперименты и возможность завершать работу при первой ошибке. Суть в том, что необходимо хорошо понимать, какие сервисы действительно доступны. Основополагающие инфраструктурные сервисы, которые когда-то были основными факторами развития облака, стали настолько продвинутыми, что решение об их использовании нередко принимается автоматически. Настройка целевой среды с определенными сетевыми адресами, подсетями, группами безопасности и маршрутами выполняется с IaC с использованием согласованной и утвержденной модели. Однако есть еще несколько важных основополагающих соображений, особенно по поводу операционных систем.

Поднимемся выше по стеку сервисов. Именно на этом уровне начинают проявляться отличия между общедоступными облачными поставщиками. Предложения варьируются в зависимости от поставщика и могут включать в себя все что угодно, от управляемых платформ баз данных до полностью обученных нейронных сетей для распознавания лиц. Понимание того, как эти сервисы вписываются в стратегию cloud native, как оценивать их для масштабного использования и существуют ли альтернативы с более узкой нишевой функцией для удовлетворения потребностей архитектуры, имеет решающее значение на протяжении всего пути перехода в облако.

Облачные сервисы: поставщик или самоуправление?

В период зарождения облачной среды базовых инфраструктурных сервисов было достаточно для создания там новых рабочих нагрузок. Однако по мере того, как эти сервисы развивались, а поставщики облачных услуг ускоряли темпы внедрения инноваций, началась разработка других сервисов, которые представляли собой управляемые версии существующих программных продуктов, используемых клиентами. Причина этого проста: цель крупных облачных провайдеров состоит в том, чтобы вводить новшества от имени клиента для сокращения недифференцированной рутинной работы в средах управления, высвобождая время и ресурсы для бизнес-потребностей.

Одной из первых областей, где этот подход был успешно реализован, стали базы данных. У крупных поставщиков облачных решений имеются управляемые версии популярных платформ баз данных для пользователей. Но должен ли клиент использовать версию, управляемую облаком, или самостоятельно развертывать

ее и управлять ею? Ответ на этот вопрос в действительности сводится к тому, специализируется ли организация на этих технологиях, и уверено ли ее руководство в том, что их самостоятельное внедрение будет дешевле и быстрее, чем при использовании услуг облачного поставщика. Даже в этом случае компании, которые решили самостоятельно управлять программным обеспечением, должны обеспечивать высокую скорость итераций, чтобы не отставать от новейшего программного обеспечения для этого сервиса, в противном случае они проиграют, когда поставщик облачных услуг будет обновлять сервис.

Подход самоуправления

Создание баз данных, иногда с применением кластеризации или других методов обеспечения высокой доступности, а также с учетом выбора дисков правильного типа и подходящих сетевых ресурсов, может быть непростой задачей. В течение многих лет этим занимались специалисты, известные как администраторы баз данных (database administrators, DBA), и корректное выполнение их обязанностей требовало много опыта и очень высокой квалификации. Облако не изменило этого, и те же самые администраторы необходимы для установки, конфигурации баз данных, а также управления платформами. На заре существования облачной технологии этот подход применялся, когда компании перемещали рабочие нагрузки в облако или разрабатывали их там, самостоятельно управляя своим программным обеспечением. С помощью виртуальных инстансов, программного обеспечения ISV-поставщиков и опытных специалистов большую часть рабочих нагрузок можно выполнять в облаке точно так же, как и локально.

Использование пакетов с открытым исходным кодом и доступ к новейшим технологиям — это одни из причин, по которым компании все еще могут сами управлять своим программным обеспечением. Поставщики облачных технологий быстро внедряют инновации, но им все же требуется некоторое время, чтобы получить готовую к выпуску версию новой технологии, которая у всех на слуху, поэтому, чтобы сократить время выхода на рынок с этой технологией, компании решают заниматься этим самостоятельно. Другая причина, по которой компании делают это, — их уверенность в своей компетентности в конкретном пакете или методике, благодаря чему предпочтение отдается самостоятельной настройке этого программного обеспечения и управлению им. Чем сложнее или важнее программный пакет, тем тяжелее и дороже будут ресурсы для управления им.

Управляемые облачные сервисы

Управляемые поставщиками облачные сервисы предоставляют клиентам технологии, в которых они нуждаются, избавляя от рутинной работы при эксплуатации облачных систем. Поставщики облачных услуг тратят больше времени и ресурсов на разработку сервисов, которые имитируют или превосходят средства управления

аналогичными технологиями на местах. Например, у поставщиков облачных услуг есть сервисы СУБД, которые предлагают те же популярные платформы баз данных, но автоматически настраиваются на охват центров обработки данных, создание инкрементных резервных копий, настройку самих себя и даже восстановление после отказа или самовосстановление в случае сбоя. Сервисы этого типа обычно могут запускаться автоматически в огромных количествах и в любой момент времени в зависимости от нужд клиентов (операционная система, исправления программного обеспечения, резервное копирование и т. д.).

По мере развития поставщики облачных услуг предлагают все больше и больше управляемых сервисов, и клиенты должны учитывать их по мере появления. Продвинутые организации cloud native будут пытаться использовать как можно больше предложений, управляемых поставщиком или связанных с бессерверными решениями, чтобы сосредоточиться на увеличении ценности бизнеса.

Зависимость от поставщика

На протяжении всей истории информационных технологий наблюдается общая для поставщиков ПО тенденция «привязывать» клиентов к своему продукту так, чтобы им было сложно от него отказаться. Несмотря на то что эта тенденция, по-видимому, реализуется для поддержки требований клиентов, часто это затрудняет, если не делает невозможным переход на новую технологию, привязывая клиентов к поставщику, которого они первоначально выбрали. Лучший пример такого поведения — старая гвардия поставщиков баз данных, использующих проприетарный код и хранимые процедуры, которые при больших базах данных могут потребовать значительных инвестиций и оказаться критичными для приложения. Инновации в приложениях такого типа хороши, когда они развиваются, позволяя клиентам пользоваться новыми функциями, как только те будут выпущены этим поставщиком. Но что произойдет, если поставщик замедлит развитие продукта или сменит платформу, а инновации перестанут поспевать за потребностями клиентов? Инвестиции в хранимые процедуры и код, написанный в соответствии со спецификацией конкретной базы данных, не дадут клиенту перейти на более динамичную и инновационную платформу.

Облачный подход в этом отношении кардинально отличается. Старый способ, при котором ИТ-центр являлся основной областью затрат, а настройка пакетов программного обеспечения и управление ими выполнялись с использованием серверов и оборудования компании, более не актуален. Компании cloud native осознали, что старая модель приведет к коммерциализации только их предложений и не позволит дифференцировать их на рынке. Единственный способ предотвратить это — избавиться от рутинной работы по управлению ИТ-сервисами и сосредоточить ресурсы на бизнес-требованиях. Облачные поставщики постоянно внедряют инновации в ряде областей и не сбавляют обороты. Они направляют значительные усилия и ресурсы на безопасность и развитие передовых сервисов, которые, как

оказалось, не могут поддерживать отдельные компании. Организации cloud native используют эти управляемые облачные сервисы, и, хотя некоторые люди считают, что это создает зависимость от определенного облачного поставщика, продвинутые компании понимают, что это новый способ ведения бизнеса.

Даже при таком новом мышлении организации должны принимать определенные меры, чтобы при необходимости иметь возможность сотрудничать с другими поставщиками облачных вычислений.

- *Наличие плана выхода.* Каждое критически важное решение требует плана выхода или миграции, если возникнет такая необходимость. Знайте, какие приложения возможно быстро переместить, какие данные с ними связаны и как это делается.
- *Разработка слабосвязанных и контейнерных приложений.* Если придерживаться 12-факторного подхода и других принципов дизайна cloud native, проектирование слабосвязанных приложений позволит упростить разработку и ограничить масштаб потенциальных проблем.
- *Тщательная организация и управление критически важными данными.* Точно знайте, какие данные у вас есть, где они находятся, а также каковы требования к их классификации и шифрованию. Это гарантирует, что в случае принятия решения о переносе в другое облако порядок операций с данными будет известен.
- *Максимальная автоматизация.* Использование IaC и других принципов DevOps позволит внести минимальные изменения в автоматизацию для поддержки другого облака. В компании cloud native автоматизации нужно уделять особое внимание.

Операционные системы

Сервисы базовой инфраструктуры обычно включают в себя виртуальные облачные инстансы. Для их запуска и работы требуется операционная система, а для облачной среды это часто сводится к выбору между разновидностью Linux или версией Windows. В дополнение к виртуальным инстансам контейнерам для работы требуется операционная система (хотя и в урезанной версии) для развертывания кода и фреймворков, которые их выполняют. Даже в развитых архитектурах cloud native, скорее всего, будут применяться отдельные серверы, предназначенные для выполнения различных задач и исправления нестандартных ситуаций, с которыми нельзя было бы справиться с помощью бессерверной или аналогичной технологии. Поэтому в ближайшем будущем операционные системы по-прежнему будут использоваться в облаке, и их выбор имеет большое значение.

Windows или Linux?

Что выбрать: Windows или Linux — это вопрос, с которым большинство клиентов сталкивается в тот или иной момент. Однако при любом масштабном развертывании

Windows и Linux (их разные версии и семейства) обычно используются совместно. Поэтому важно знать, какие версии необходимы, а какие предлагает поставщик облачных услуг. Как правило, у поставщика облачных вычислений есть множество различных вариантов каждой операционной системы. Одни потребуют расходов на лицензирование, другие будут бесплатными. Стоимость операционных систем может составлять значительную часть расходов, поэтому попытка свести к минимуму их использование или работа только с бесплатными рекомендуется для клиентов, которые действуют исключительно в облачной среде.

Действительно ли операционные системы имеют значение?

Действительно ли важны операционные системы? И да и нет. Да, потому что контейнеры — по-прежнему распространенная технология для разработки микросервисов, для которой требуется операционная система. Нет, потому что по мере развития архитектуры все больше управляемых облачных сервисов будут использоваться без серверов. Поскольку управление работой операционных систем может требовать все больше и больше усилий, продвинутые компании cloud native ограничиваются использованием устаревших приложений и сред, которые по-прежнему требуют установки определенных библиотек.

Поскольку облако представляет собой управляемую API среду, определение операционных систем несколько меняется. Конечно, виртуальные инстансы и контейнеры все еще нуждаются в операционных системах. Однако сама полномасштабная облачная среда превращается в своего рода операционную систему. Микросервисы, события, контейнеры и управляемые поставщиком сервисы теперь взаимодействуют через вызовы API сервисов, хранилища объектов и другие механизмы разделения компонентов. Тенденция представлять облако и рабочие нагрузки, развернутые в нем, как крупномасштабную операционную систему будет все более популярной.

Резюме

В данной главе мы узнали ответы на вопросы, которые могут возникнуть при выборе технологического стека. В следующей главе речь пойдет о масштабируемости и доступности в облачных архитектурах.

5 Масштабируемость и доступность

В предыдущих главах мы определили, что такое архитектура cloud native и как она влияет на людей, процессы и технологии. Путь к созданию облачных систем непростой и длительный. Могут потребоваться годы на то, чтобы полностью реализовать потенциал перехода к технологиям cloud native по мере развития вашей организации и ее культуры. Набор навыков и подходы к решению проблем должны меняться со временем. Допускаются ошибки, извлекаются уроки, и системы развиваются. Мы определили общие критерии процесса перехода в облако. Переход затронет ваш бизнес, людей, управление, безопасность платформы и эксплуатацию. Каждое приложение и каждая компания могут соответствовать различным уровням зрелости в зависимости от того, на каком этапе перехода они находятся. Чтобы вы лучше понимали этот процесс, мы создали модель CNMM (модель зрелости с точки зрения облачной ориентированности). Эта модель позволяет понять, на каком этапе в данный момент находятся ваши организация, стек приложений или система и какие функции им могут понадобиться, чтобы стать более облачными. Кроме того, мы описали жизненный *цикл разработки программного обеспечения (software development life cycle, SDLC)* для приложений cloud native и узнали, как выбрать лучший технологический стек, который подходит для конкретной компании.

В следующих четырех главах мы попытаемся определить основные принципы проектирования cloud native. Эти архитектурные принципы являются ключевыми характеристиками, которые делают систему облачно-ориентированной. Сами по себе они не являются уникальными для облачных приложений или систем. Однако вместе они представляют собой набор возможностей, которые не может предложить ни одна платформа, кроме облака. Мы представим эти компоненты по одному в каждой из следующих глав. Для каждого компонента опишем основные принципы проектирования, чтобы определить архитектуру и развертывание. Мы рассмотрим примеры применения этих принципов к различным уровням стека приложений. А еще обсудим инструменты, проекты с открытым исходным кодом, сервисы cloud native и сторонние программные решения, которые могут помочь в достижении этих целей.

В данной главе поможем читателю понять, с какими масштабами работают современные облачные провайдеры. В настоящий момент наблюдается явный переход от традиционного масштаба центров обработки данных, с которым знакомы большинство опытных ИТ-специалистов, к *гипермасштабам*. По сути, гипермасштабная

облачная инфраструктура порождает многие из концепций и подходов, обсуждаемых здесь. Мы также представим основные характеристики масштабируемой и доступной облачной системы, что поможет читателю принимать обоснованные архитектурные решения. Наконец, обсудим доступные на данный момент инструменты, с помощью которых создается инфраструктура самовосстановления, составляющая основу устойчивых архитектур cloud native.

В этой главе будут рассмотрены следующие темы.

- Глобальная облачная инфраструктура и общая терминология.
- Концепции облачной инфраструктуры (регионы и зоны доступности).
- Автомасштабирование групп и балансировщики нагрузки.
- Стратегии определения размеров виртуальных машин.
- Постоянно работающие архитектуры.
- Проектирование сетевого резервирования и основных сервисов.
- Мониторинг.
- Инфраструктура как код.
- Неизменяемые развертывания.
- Самовосстанавливающаяся инфраструктура.
- Основные принципы масштабируемой и доступной архитектуры.
- Сервис-ориентированные архитектуры.
- Набор инструментов cloud native для масштабируемой и доступной архитектуры.

Введение в гипермасштабную облачную инфраструктуру

При развертывании систем или стеков в облаке важно осознавать масштаб, в котором работают ведущие поставщики облачных услуг. Три крупнейших облачных провайдера создали зону обработки данных, охватывающую почти весь мир. Для обеспечения минимальной задержки и высокой пропускной способности подключения к системам, работающим в глобально развернутых центрах обработки данных, они соединили разные уголки мира широкополосными оптоволоконными кабелями. Масштабы, в которых работают эти три облачных провайдера, настолько превосходят масштабы других игроков, что индустрии пришлось ввести новое обозначение — гипероблако. На рис. 5.1 показана глобальная зона обслуживания AWS — крупнейшего по общей вычислительной мощности облачного провайдера (по оценкам Gartner) (https://aws.amazon.com/about-aws/global-infrastructure/).

Рис. 5.1

В дополнение к парку построенных и работающих центров обработки данных гипероблачные провайдеры развернули широкополосные трансконтинентальные оптоволоконные сети для эффективного управления трафиком между каждым из своих кластеров центров обработки данных. Это позволяет клиентам соответствующих платформ создавать распределенные приложения в нескольких регионах с очень низкой задержкой и стоимостью.

Теперь давайте посмотрим на изображение, которое показывает глобальный центр обработки данных AWS и инфраструктуру ее магистральных сетей (компания AWS впервые публично поделилась своими глобальными схемами на ежегодной конференции разработчиков Re: Invent в ноябре 2016 года, где ее представлял выдающийся инженер и вице-президент AWS Джеймс Гамильтон) (рис. 5.2).

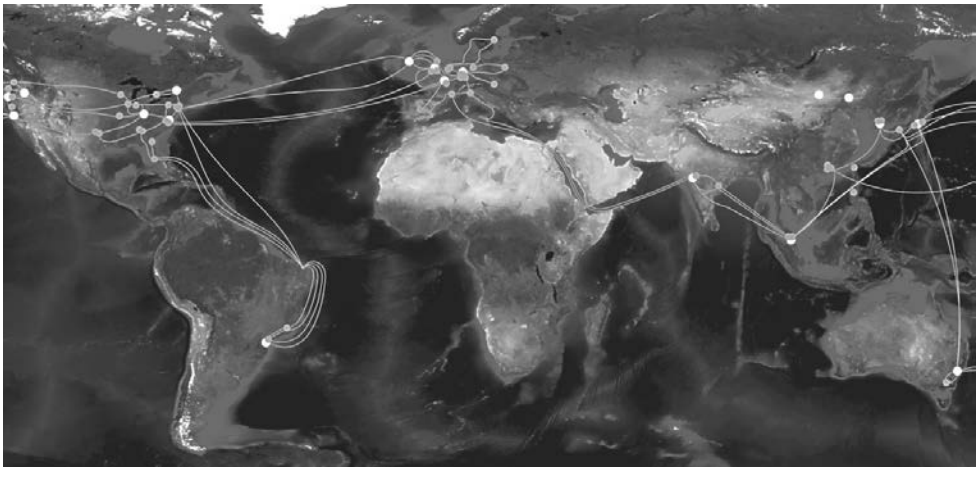


Рис. 5.2

Google Cloud Platform (GCP) и Microsoft Azure обеспечивают схожий географический охват. Все три гипероблачные платформы предоставляют базовые облачные сервисы в абстрактной инфраструктуре, называемой *регионом (region)*. Регион — это совокупность центров обработки данных, работающих вместе с соблюдением строгих требований к задержке и производительности. Такая архитектура позволяет пользователям облачной платформы распределять свои рабочие нагрузки по нескольким центрам обработки данных, составляющим регион. AWS называет это *зонами доступности (availability zones, AZ)*, а GCP — просто *зонами*.

Практические рекомендации для архитектуры cloud native

Распределите рабочие нагрузки по нескольким зонам в пределах одного региона, чтобы обеспечить высокую доступность стека и его устойчивость к сбоям оборудования или компонентов приложения. Зачастую это не требует дополнительных затрат, но дает стекам явное операционное преимущество за счет распределения вычислительных узлов по нескольким изолированным центрам обработки данных.

На схеме на рис. 5.3 показаны три разные зоны доступности в одном регионе.

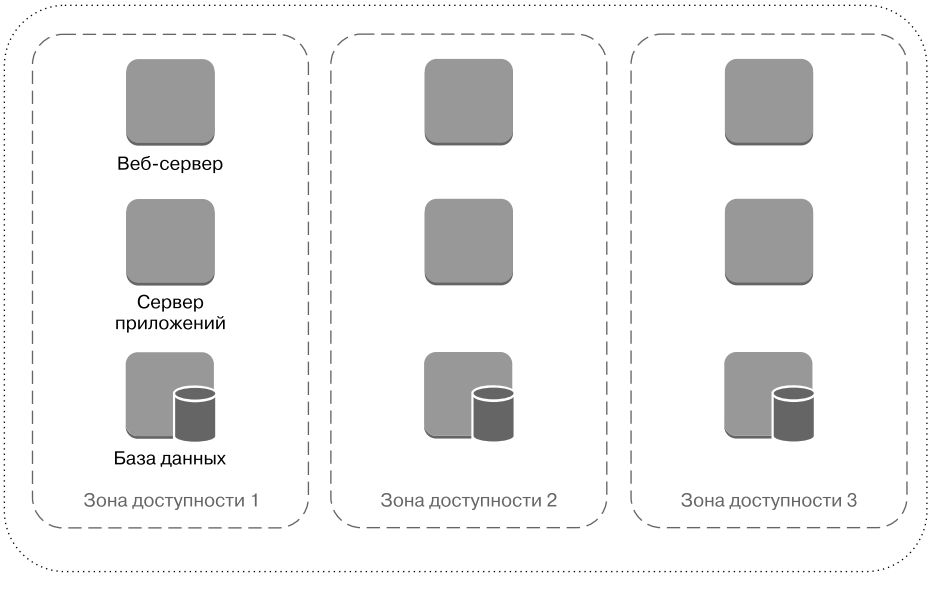


Рис. 5.3

Как показано на схеме, рассредоточение вычислительных рабочих нагрузок по нескольким зонам доступности в пределах региона уменьшает масштаб потенциального ущерба в случае прерывания обслуживания, будь то сбой приложения или машины. Обычно это не приводит к увеличению затрат, поскольку зоны доступ-

ности специально созданы для обеспечения высокой производительности и возможности расширения в пределах региона.

Ключевая концепция, которую необходимо понять при проектировании облачных систем, — масштаб потенциального ущерба. Мы определяем его, оценивая *приложения или вспомогательные системы, на которые может повлиять отказ основного компонента проектирования*. Это понятие может быть применено и к центру обработки данных, и к отдельному микросервису. Нужно понять зависимости в собственной архитектуре, оценить свою склонность к риску, определить количественные последствия минимизации масштаба потенциального ущерба и спроектировать стек в соответствии с этими параметрами.

Ключевым элементом минимизации последствий потенциальных сбоев в облаке является эффективное распределение систем по регионам и зонам. Крупнейшие провайдеры предлагают ряд сервисов, которые помогают архитекторам делать это эффективно, — балансировщики нагрузки и группы автоматического масштабирования.

Балансировщики нагрузки (load balancers, LB) — не уникальная разработка для облака, но все основные платформы имеют собственные сервисы, которые предоставляют этот функционал. Эти нативные сервисы, работающие на многих машинах, обеспечивают гораздо более высокий уровень доступности, чем виртуальный балансировщик нагрузки, который работает на виртуальной машине, управляемой потребителем облака.

Практические рекомендации для архитектуры cloud native

По возможности задействуйте облачные балансировщики нагрузки. Это позволяет передать обслуживание, которым прежде занимался потребитель, поставщику облачных услуг. Пользователю облака больше не нужно беспокоиться о поддержании работоспособности балансировщика нагрузки, поскольку этим будет управлять CSP. Балансировщик нагрузки работает с несколькими контейнерами или парками виртуальных машин, а это означает, что аппаратные или программные сбои легко устраняются в фоновом режиме незаметно для пользователя.

Концепция балансировщиков нагрузки должна быть знакома всем, кто работает в ИТ-индустрии. Наряду с балансировкой нагрузки облачные сервисы позволяют предоставлять пользователям услуги DNS. Эта комбинация дает возможность клиентам создавать глобально доступные приложения, которые легко распространяются на выбранные ими географические области. Такие сервисы, как Amazon Route53 на платформе AWS, позволяют пользователям проектировать правила маршрутизации на основе задержек и географических ограничений для подключения конечных потребителей к наиболее производительному стеку (или для предотвращения доступности этих сервисов в зависимости от местоположения конечных пользователей). Примером этого может быть ограничение доступа к вашему приложению для жителей отдельных стран в целях соблюдения действующих законов о санкциях.

Практические рекомендации для архитектуры cloud native

Используйте облачные сервисы системы доменных имен (DNS) (AWS Route53, Azure DNS или GCP Cloud DNS). Эти сервисы изначально интегрированы с сервисами балансировки нагрузки на каждой платформе. Задействуйте политики маршрутизации, такие как маршрутизация на основе задержки (LTR), для создания глобально доступных приложений, работающих в разных регионах. Используйте такие функции, как GeoDNS, для маршрутизации запросов от определенных географических регионов (или стран) к конкретным конечным точкам.

Другим важным инструментом в наборе инструментов, предоставляемых облаком, является развертывание *групп автоматического масштабирования (auto scaling groups, ASG)* — новая сервисная абстракция, которая позволяет пользователям динамически реплицировать и расширять виртуальные машины с приложениями на основе различных сигналов тревоги или флагов. Для развертывания ASG необходимо предварительно настроить и сохранить стандартизированный образ приложения. ASG почти всегда должны быть связаны с балансировщиками нагрузки, чтобы с ними можно было эффективно работать, поскольку трафик от потребителя приложения должен быть грамотно направлен к доступному и производительному вычислительному узлу в парке ASG. Это можно сделать несколькими способами, включая балансировку по кругу или развертывание системы очередей. Базовая конфигурация автоматического масштабирования для двух AZ показана на рис. 5.4.

Группа автоматического масштабирования на схеме настроена минимум на две виртуальные машины, по одной в каждой зоне доступности (*веб-сервер 1* и *веб-сервер 2*). Трафик к этим узлам показан черным цветом. По мере того как все больше пользователей получают доступ к приложению, ASG реагирует развертыванием большего количества серверов для ASG в нескольких AZ (*веб-сервер 3* и *веб-сервер 4*), при этом дополнительный трафик отображается серым цветом.

Мы ввели критически важные элементы отказоустойчивой облачной системы: балансировку нагрузки, автоматическое масштабирование, развертывание в нескольких регионах и глобальный DNS. Функцию автоматического масштабирования этого стека можно имитировать за счет гибкого расширения и сокращения количества контейнеров.

Практические рекомендации для архитектуры cloud native

При проектировании стека рентабельнее использовать с балансировщиком нагрузки множество небольших виртуальных машин. Это повышает степень детализации затрат и увеличивает степень резервирования в системе в целом. Рекомендуется также создавать архитектуру без сохранения состояния, поскольку это устраняет зависимость приложения от одной виртуальной машины, что значительно упрощает восстановление сеанса в случае сбоя.

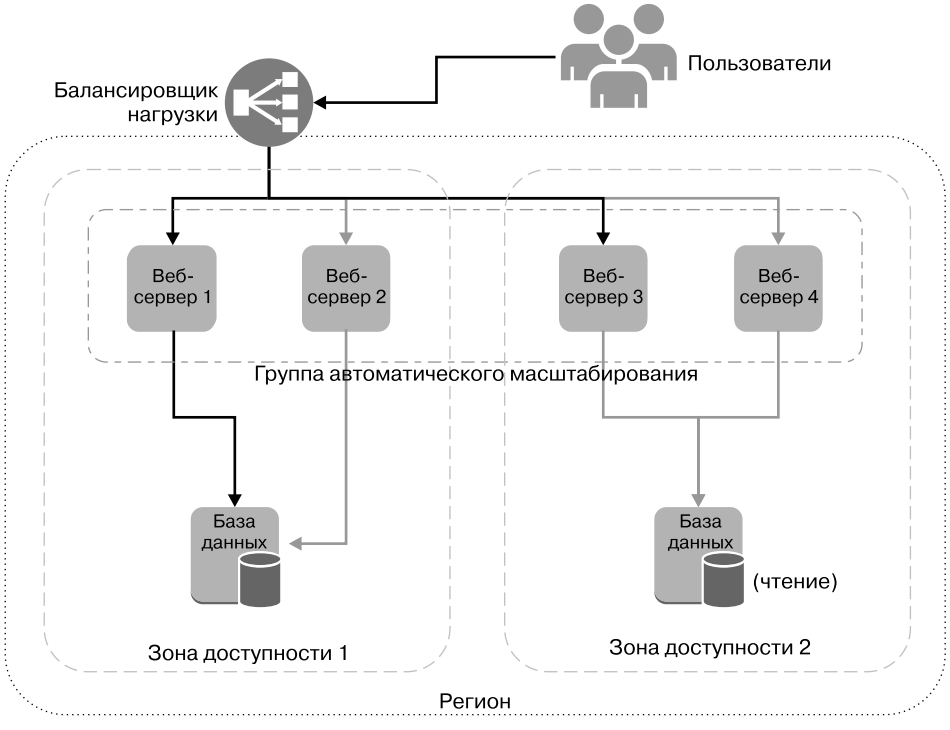


Рис. 5.4

Сравним группы автоматического масштабирования с виртуальными машинами разных размеров (рис. 5.5).

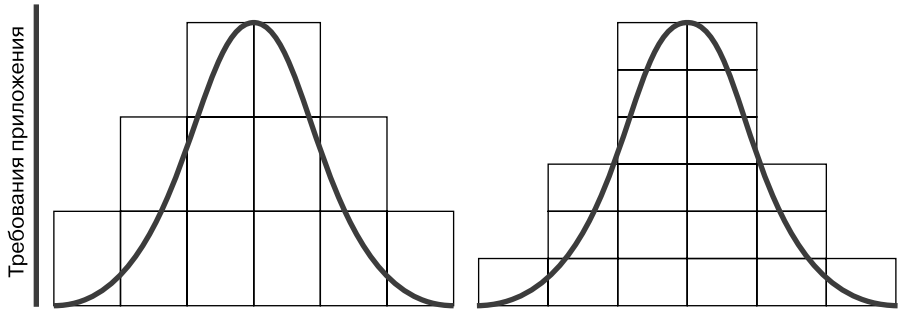


Рис. 5.5

Здесь показано преимущество создания групп автоматического масштабирования с более мелкими виртуальными машинами. Линия на графике представляет требования или трафик данного приложения. Каждый из блоков обозначает одну

виртуальную машину. Виртуальные машины в группе автоматического масштабирования слева имеют больше памяти, вычислительных ресурсов и, как следствие, более высокую почасовую оплату. Виртуальные машины справа имеют меньше памяти и вычислительной мощности, из-за чего почасовая оплата ниже.

На схеме продемонстрировано преимущество использования в парке ASG меньших вычислительных узлов. Любые пробелы в блоке, которые появляются над синей линией, — это потраченные впустую оплаченные ресурсы. Что означает: производительность системы не идеально настроена под требования приложения, в результате ресурсы *простаивают*. Минимизируя спецификации виртуальных машин, можно добиться значительной экономии средств. Кроме того, за счет уменьшения размера виртуальных машин стек приложений становится более распределенным. Сбой одной виртуальной машины или контейнера не поставит под угрозу общее состояние стека, поскольку в группе есть свободные виртуальные машины, на которые можно переключиться.

Применение сервисов балансировки нагрузки с автоматическими группами масштабирования, динамической маршрутизации, архитектур без сохранения состояния, использующих легкодоступные и производительные сервисы СУБД, распределенные по нескольким зонам (или группам центров обработки данных), говорит о зрелости архитектуры cloud native.

Постоянно работающие архитектуры

В течение многих лет у архитекторов системы было два главных направления работы: доступность системы и *возможность ее восстановления*, часто называемая аварийным восстановлением. Эти две концепции предназначены для описания характеристик, присущих системе, развернутой в ограниченной локальной инфраструктуре. В такой инфраструктуре имеется ограниченное количество физических или виртуальных ресурсов, выполняющих очень специфические функции или поддерживающих конкретное приложение. Данные приложения разработаны таким образом, что исключена возможность их распределенного запуска на нескольких машинах. Эта парадигма означает, что вся система имеет много отдельных точек отказа, будь то отдельный сетевой интерфейс, виртуальная машина, физический сервер, виртуальный диск или том и т. д.

Учитывая эти изначальные недостатки, архитекторы предусмотрели два принципиальных параметра оценки эффективности системы. Способность системы продолжать работать и выполнять свои функции известна как доступность. В случае сбоя системы ее способность к восстановлению определяется двумя показателями.

- *Допустимое время восстановления (recovery time objective, RTO)* — время, необходимое для приведения системы в приемлемое функциональное состояние после сбоя.

- *Допустимая точка восстановления (recovery point objective, RPO)* — максимальная продолжительность времени, за которое могут быть потеряны данные во время простоя.

Если взять архитектуру, полностью ориентированную на облако, то есть несколько важных факторов, влияющих на эти старые парадигмы и позволяющих их расширить.

- *Облако предоставляет архитекторам бесконечное количество вычислений и памяти для решения всех практических задач*. Гипермасштабируемость ведущих облачных провайдеров, которую мы обсуждали ранее в этой главе, демонстрирует размер и область применения современных систем.
- *Сервисы, предоставляемые облачными платформами, по своей природе отказоустойчивы и гибки*. Потребление или интеграция этих сервисов в стек обеспечивает уровень доступности, которого редко можно достичь при попытке создать легкодоступный стек с использованием собственных инструментов. Например, AWS *Simple Storage Service (S3)* — это отказоустойчивый и легкодоступный сервис хранения объектов, который обеспечивает надежность 99,99999999 % (то есть вероятность того, что объект будет потерян навсегда, ничтожно мала). Доступность *Service Level Agreement (SLA)* для стандарта S3 составляет 99,99 %. Объекты, которые вы загружаете на S3, автоматически реплицируются в двух других зонах доступности в регионе для резервирования без дополнительных затрат для пользователя. Он просто создает логическое хранилище, называемое *корзиной*, и загружает туда свои объекты для последующего поиска. Обслуживание физических и виртуальных машин и контроль над ними осуществляются AWS. Чтобы попытаться приблизиться к уровню доступности и устойчивости, обеспечиваемому AWS, в вашей собственной системе, потребуется много времени, инженерных ресурсов и финансовых вложений.
- *Доступны собственные сервисы и возможности, которые помогают архитекторам и разработчикам в мониторинге, маркировке и выполнении действий для поддержания работоспособности системы*. Например, сервис мониторинга Cloudwatch от AWS собирает метрики производительности и работоспособности виртуальных машин в стеке. Метрики Cloudwatch можно применять в сочетании с группами автомасштабирования или функцией восстановления инстансов, чтобы система могла автоматически реагировать на определенные уровни производительности. Используя эти службы мониторинга совместно с бессерверными функциями, архитекторы могут создать среду, в которой система будет автоматически реагировать на различные показатели.

Учитывая эти облачные функции, мы полагаем, что этот новый подход к облачным архитектурам позволяет реализовать парадигму *постоянной работы*. Она помогает планировать сбои и проектировать архитектуру таким образом, чтобы система могла восстановиться и откорректировать свою работу без какого-либо вмешательства пользователя. Этот уровень автоматизации говорит о высоком уровне развития системы и имеет самый высокий показатель по шкале CNMM.

Важно отметить, что в любом виде деятельности люди рано или поздно терпят неудачу, сталкиваются с трудностями или останавливаются на полпути, и облако не исключение. Поскольку мы не обладаем даром предвидения и постоянно развиваем наши навыки в области ИТ, в определенный момент что-то *неизбежно* сломается. Понимание и принятие этого факта лежит в основе парадигмы постоянной работы — планирование сбоев является единственным гарантированным способом их смягчения или предотвращения.

Постоянная работа: ключевые архитектурные элементы

Существуют определяющие функции архитектур cloud native, которые позволяют поддерживать всегда работающую технически устойчивую архитектуру. Вопрос не стоит ребром — «все или ничего». Многие системы сочетают в себе разные возможности, о которых дальше пойдет речь. Не терзайте себя и своих системных архитекторов мыслями о том, что все это можно внедрить в ваш проект в одночасье. Для реализации ключевых архитектурных элементов, которые мы будем обсуждать, потребуется постепенный, эволюционный подход. Как говорит Вернер Фогельс (технический директор Amazon), «нет таких систем, которые не ломаются». Если мы запланируем неизбежный сбой, то сможем спроектировать системы таким образом, чтобы избежать этого.

Сетевая избыточность

Подключение как к облаку, так и во всех ваших средах должно быть выполнено с высокой степенью доступности и избыточности. В облачном мире из этого следуют два основных вывода. Первым следствием является физическое подключение из локальной среды или клиента к облаку. Все гипероблачные поставщики предоставляют частное высокоскоростное сетевое соединение вместе с партнерами-провайдерами (например, AWS Direct Connect, Azure ExpressRoute и GCP Cloud Interconnect). Рекомендуется резервировать основное широкополосное подключение с помощью функции переключения при отказе (другое частное подключение или VPN-туннель через Интернет) с использованием отдельных физических каналов.

Практические рекомендации для архитектуры cloud native

Для обеспечения избыточных сетевых подключениях в облачной среде задействуйте два разных изолированных сетевых соединения. Это особенно важно для предприятий, которые полагаются на свои облачные среды в отношении критически важных приложений. В случае небольших или менее критических развертываний для облегченных сред можно рассмотреть или комбинацию частного оптоволоконного канала и VPN, или два канала VPN.

Вторым следствием является избыточность физических и виртуальных конечных точек как в облачной, так и в локальной среде. Это спорный вопрос для облачных конечных точек, поскольку каждая из них предоставляется в конфигурации *«шлюз как услуга»*. Эти виртуальные шлюзы работают на гипервизоре облачных провайдеров на нескольких физических машинах в нескольких центрах обработки данных. Часто упускают из виду, что облачно-ориентированный подход должен распространяться не только на само облако, но и на конечные точки клиента. Это необходимо для того, чтобы обеспечить отсутствие единой точки отказа, которая ухудшала бы производительность или доступность системы для конечных пользователей. Таким образом, подход cloud native распространяется и на локальную сеть потребителя. Он требует применения нескольких избыточных устройств с параллельными сетевыми подключениями, подобно тому как облачные провайдеры создают подключения к своим физическим центрам обработки данных.

Практические рекомендации для архитектуры cloud native

Для предприятий и критически важных бизнес-сред используйте два отдельных физических устройства на стороне клиента (локально) в качестве конечной точки облачного подключения. Проектирование системы для непрерывной работы в облаке бессмысленно, если вашему подключению к среде угрожает единая аппаратная точка отказа на стороне клиента.

Посмотрим на схему (рис. 5.6), показывающую связь между облачным провайдером и корпоративным центром обработки данных.

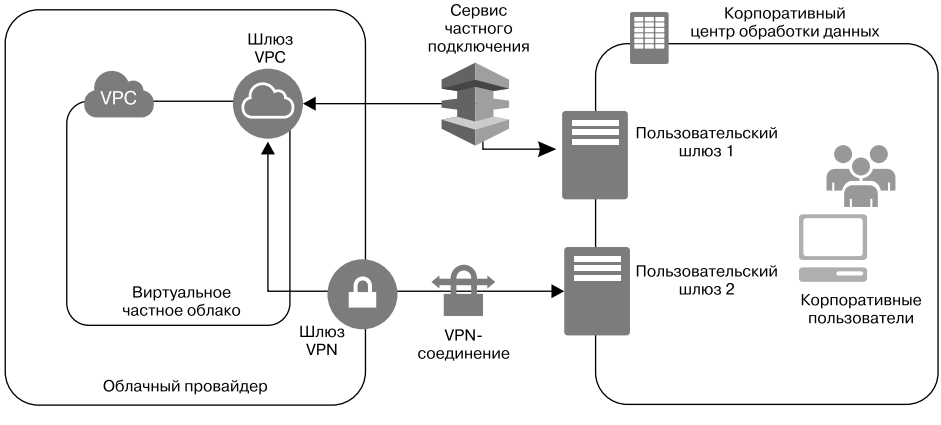


Рис. 5.6

При подключении из корпоративного центра обработки данных, офиса или с сайта клиента к облачной среде используйте резервные маршруты подключения для поддержания доступности облачной среды. В случае отказа устройства, сбоя физического соединения или прерывания обслуживания всегда должен быть доступен

резервный вариант. Это может быть сочетание дорогостоящей высокопроизводительной основной линии и недорогой низкопроизводительной резервной линии (то есть VPN через Интернет).

Резервирование основных сервисов

Резервирование (обеспечение избыточности) ключевых сервисов и приложений, используемых остальным ИТ-стеком, — еще один важный аспект проектирования облачных архитектур. На предприятии имеется множество устаревших приложений, которые только начинают адаптироваться к модели cloud native. При переходе в облако очень важно реструктурировать или реорганизовать эти приложения, обеспечив их надежную работу в новой среде.

В качестве примера можно назвать *Active Directory (AD)* — критический компонент для обеспечения продуктивности почти в любой организации. AD поддерживает и предоставляет доступ к учетным записям, компьютерам и сервисам путем аутентификации и авторизации их действий.

Существуют различные уровни зрелости cloud native постоянных сервисов AD. На одном конце спектра компании просто расширяют свою сеть до облака и используют инфраструктуру AD, которая уже существует локально. Такой подход наименее производителен и сводит к минимуму преимущества, которые может предложить облако. В более продвинутой схеме архитекторы расширяют лес до облака путем развертывания *контроллеров домена (domain controllers, DC)* или *контроллеров домена только для чтения (read-only domain controllers, RODC)* в облачной сетевой среде. Это обеспечивает более высокий уровень избыточности и производительности. Для действительно облачного развертывания AD ведущие облачные платформы предоставляют собственные сервисы, поддерживающие полностью управляемые, крупномасштабные развертывания AD, обслуживание которых обходится намного дешевле по сравнению с собственной физической или виртуальной инфраструктурой. AWS имеет несколько вариантов (сервис каталогов AWS, управляемая Microsoft AD и AD Connector), Microsoft предоставляет Azure AD, а в облаке Google есть синхронизация каталогов. На рис. 5.7 представлено три различных параметра активной идентификации, от минимального *(вариант 1)* до наиболее производительного *(вариант 3)*, в зависимости от того, где размещены контроллеры домена. Здесь же показан диапазон развертывания AD.

В варианте 1 мы полагаемся на подключение к локальному серверу AD для аутентификации и авторизации. Вариант 2 — развернуть DC или RODC в облачной среде. Наиболее облачный вариант 3 состоит в том, чтобы перенести всю *недифференцированную рутинную работу* в сервис cloud native. Облачный сервис разворачивает инфраструктуру AD и управляет ею, освобождая пользователя от второстепенных задач. Этот вариант дает намного лучшие производительность и доступность. При использовании этого подхода AD автоматически развертывается в нескольких зонах доступности, с помощью нескольких щелчков кнопкой

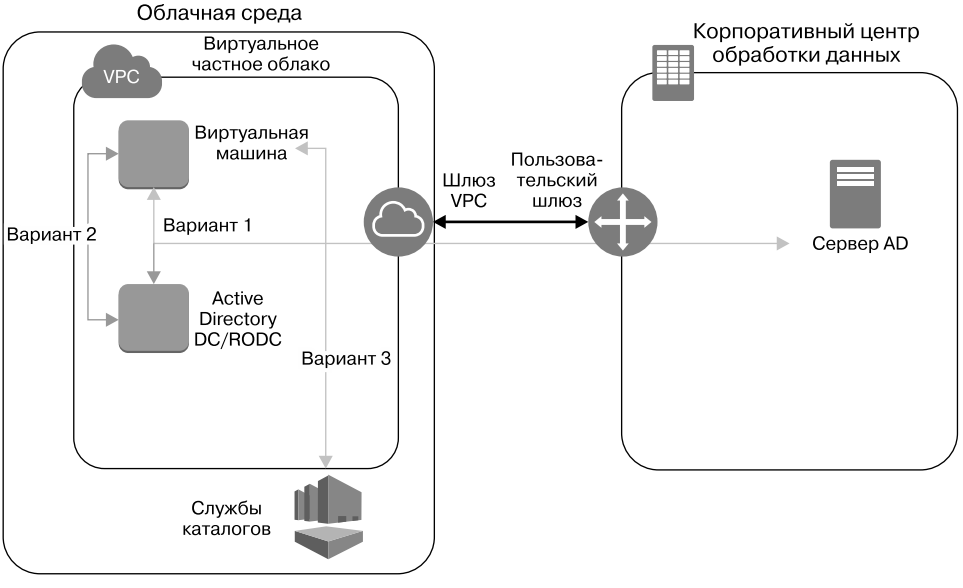


Рис. 5.7

мыши обеспечивает возможность масштабирования для увеличения количества контроллеров домена и поддерживает расширенные возможности, необходимые для локального развертывания AD (например, единый вход, групповые политики и резервное копирование).

Кроме этого, облачные сервисы AD изначально интегрируются с другими облачными сервисами, что облегчает развертывание виртуальных машин. Виртуальные машины очень легко добавить в домены, а доступ к самой облачной платформе можно объединить на основе идентификационных данных и сведений о пользователях в AD. Следует также отметить, что эти облачные сервисы обычно (если не всегда) поддерживают *протокол облегченного доступа к каталогам (Lightweight Directory Access Protocol, LDAP)*, что означает применение альтернативных решений Microsoft AD, таких как OpenLDAP, Samba и Apache Directory.

Система доменных имен (Domain Name System, DNS) придерживается модели, аналогичной AD, с различными уровнями облачной активности в зависимости от модели развертывания. Для обеспечения наивысшего уровня доступности и масштабируемости используйте ведущие облачные сервисы, такие как AWS Route53, Azure DNS и Google Cloud DNS (речь о них пойдет далее). Размещенные зоны (общедоступные и частные), поддерживаемые этими внутренними сервисами, обеспечивают наилучшую производительность с учетом удельной стоимости. Другие централизованные сервисы, помимо AD и DNS, предлагают антивирусную защиту (AV), системы обнаружения и предотвращения вторжений (IPS/IDS), а также ведение журналов. Эти темы будут более подробно рассмотрены в главе 6.

Помимо основных сервисов, которые может использовать предприятие, в большинстве стеков крупномасштабных приложений необходимо использовать промежуточные программные компоненты или очереди, которые помогают управлять большими и сложными потоками трафика через стек. Как можно догадаться, есть два основных подхода, которые архитектор может применять для развертывания. Первый — управление собственной системой очередей, развернутой на виртуальных машинах облака. Пользователям облаков приходится отвечать за эксплуатационные аспекты развертывания очереди (в качестве примеров можно привести продукты от Dell Boomi или Mulesoft), включая настройку развертываний с несколькими зонами AZ для обеспечения высокой доступности.

Второй вариант — использовать сервис очереди или шины сообщений, предлагаемый облачными платформами. CSP управляет недифференцированной рутинной работой, и архитектору или пользователю остается лишь настроить и использовать сервис. Ведущими сервисами являются *Simple Queue Service (SQS)* от AWS, AmazonMQ, *Simple Notification Service (SNS)* от Amazon, Azure Storage Queues, Azure Service Bus Queues, а также GCP Task Queue.

Облачные сервисы очередей размещаются в инфраструктуре с поддержкой резервирования, обеспечивая гарантированную доставку сообщений не менее одного раза (at-least-once), некоторые поддерживают функцию *«первым пришел — первым вышел» (First In First Out, FIFO)*. Возможность масштабирования сервиса позволяет очереди сообщений давать одновременный параллельный доступ множеству производителей и потребителей. Это делает его идеальным кандидатом на то, чтобы занимать центральное место развернутой сервис-ориентированной архитектуры (SOA).

Мониторинг

Для приложений cloud native система мониторинга является эквивалентом нервной системы организма. Без нее сигналы о расширении и сокращении групп автоматического масштабирования, оптимизация производительности инфраструктуры и взаимодействие с конечным пользователем были бы невозможны. Мониторинг необходимо использовать на всех уровнях стека, чтобы максимизировать обратную связь, помочь в интеллектуальной автоматизации и проинформировать владельцев ИТ-ресурсов о том, куда следует направить усилия и средства. Мантрой для всех архитекторов и инженеров должна стать фраза: «Если вы не можете измерить это, вы не можете управлять этим».

Мониторинг в среде cloud native должен главным образом охватывать четыре области.

- *Мониторинг инфраструктуры*. Собирает и сообщает данные о производительности хостов, сети, баз данных и любых других основных облачных сервисов,

которые используются в вашем стеке (помните, что даже облачные сервисы могут выходить из строя или давать сбой, поэтому их необходимо мониторить на предмет работоспособности и производительности).

- *Мониторинг приложений.* Собирает и сообщает данные о применении локальных ресурсов, поддерживающих выполнение приложения, наряду с метриками, уникальными для вашего собственного приложения cloud native (например, временем возврата запроса).
- *Мониторинг пользователей в режиме реального времени.* Собирает и сообщает данные обо всех взаимодействиях пользователей с вашим сайтом или клиентом. Такие механизмы мониторинга нередко первыми выявляют признаки неисправностей или ухудшения качества обслуживания в системе.
- *Мониторинг журналов.* Собирает журналы со всех хостов, приложений и устройств безопасности в централизованно управляемое и расширяемое хранилище.

Практические рекомендации для архитектуры cloud native

Во всех случаях мониторинга рекомендуется по возможности вести и хранить журналы данных. Их анализ помогает создавать новые интересные модели использования, помогая определить, когда нужно предварительно масштабировать или идентифицировать проблемные или неэффективные сервисные компоненты.

Во всех четырех названных случаях облако изначально поддерживает создание, загрузку и хранение журналов. AWS Cloudwatch и CloudTrail, Azure Monitor, Google Cloud Stackdriver Monitoring и IBM Monitoring and Analytics — это примеры нативных сервисов, предоставляемых на этих платформах.

На схеме на рис. 5.8 показаны различные источники журналов, встречающихся в гибридной среде (облачной и локальной).

Источники могут быть следующие.

- *Источник 1* — протоколирование вызовов API, выполняемых ИТ-командами и машинами для изменения настроек и конфигураций среды.
- *Источник 2* — журналы конечных пользователей, генерируемые мобильными и стационарными веб-клиентами.
- *Источник 3* — журналы производительности инфраструктуры, создаваемые локальными и облачными ресурсами.
- *Источник 4* — пользовательские журналы, генерируемые вашим приложением. Они являются специфическими и считаются относящимися к общему состоянию стека.

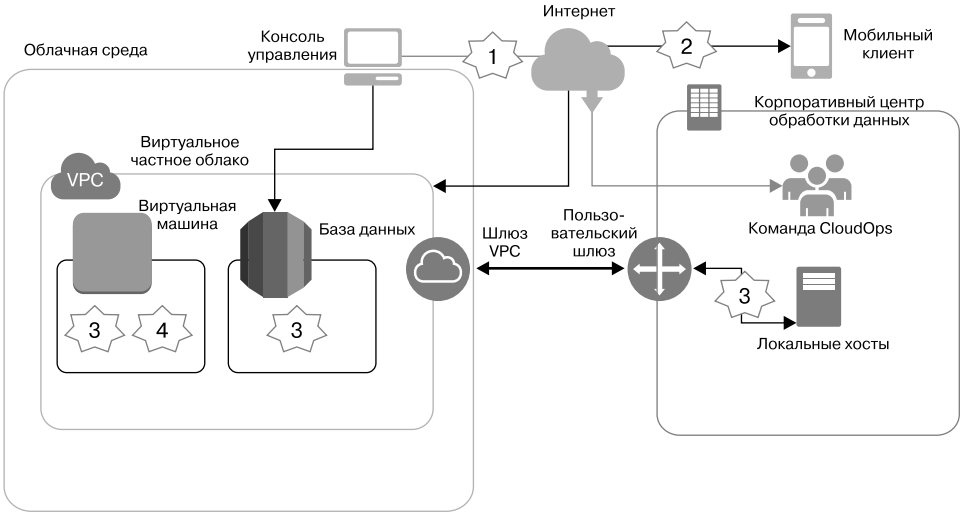


Рис. 5.8

В большинстве случаев на выбор предлагается несколько вариантов решения, начиная с собственных облачных сервисов, сторонних продуктов, развернутых и управляемых на облачных вычислительных ресурсах, и заканчивая комбинацией того и другого. Для предприятия популярными (хотя иногда и дорогими) вариантами являются Splunk, Sumo Logic, Loggly и Dynatrace. Эти сторонние решения предоставляют широкие возможности для сбора, хранения и анализа больших объемов данных, которые генерируются во всех четырех случаях использования. Стандартные API доступны для создания и передачи данных от источников непосредственно в приложения.

Вторгшись в это пространство и постоянно предоставляя все больше функций и возможностей, CSP запустили собственные сервисы регистрации и мониторинга, изначально интегрированные с другими сервисами на платформе. Примеры — AWS CloudTrail и Cloudwatch. Они предоставляют развитый сервис под ключ, который изначально позволяет вести журналы на платформе. Оба легко интегрируются с другими сервисами, предлагаемыми AWS, что повышает их ценность для пользователя.

Сервис AWS CloudTrail помогает обеспечить управление, соответствие требованиям, операционный аудит и аудит рисков в среде AWS, регистрируя каждое уникальное взаимодействие с учетной записью. Он пользуется тем, что каждое событие в учетной записи (изменение конфигурации, инициированное человеком или машиной) выполняется через API. Запросы API фиксируются, хранятся и отслеживаются через CloudTrail. Это обеспечивает командам по эксплуатации и безопасности исчерпывающее представление о том, *кто*, *что* и *когда* изменил. AWS

CloudWatch — это дополнительный сервис, который позволяет пользователям собирать и отслеживать показатели с платформы или из пользовательских источников на уровне приложений. Он разрешает пользователям создавать собственные панели мониторинга, что значительно упрощает объединение соответствующих показателей для рабочих групп и отчетов. Наконец, CloudWatch позволяет им устанавливать сигналы тревоги, которые могут автоматически инициировать события, такие как отправка SMS или уведомлений по электронной почте, когда для заданного ресурса выполняются определенные условия (например, критическая загрузка ЦП или недостаточная емкость чтения БД).

На рис. 5.9 показан пример пользовательской панели, созданной с помощью AWS CloudWatch.

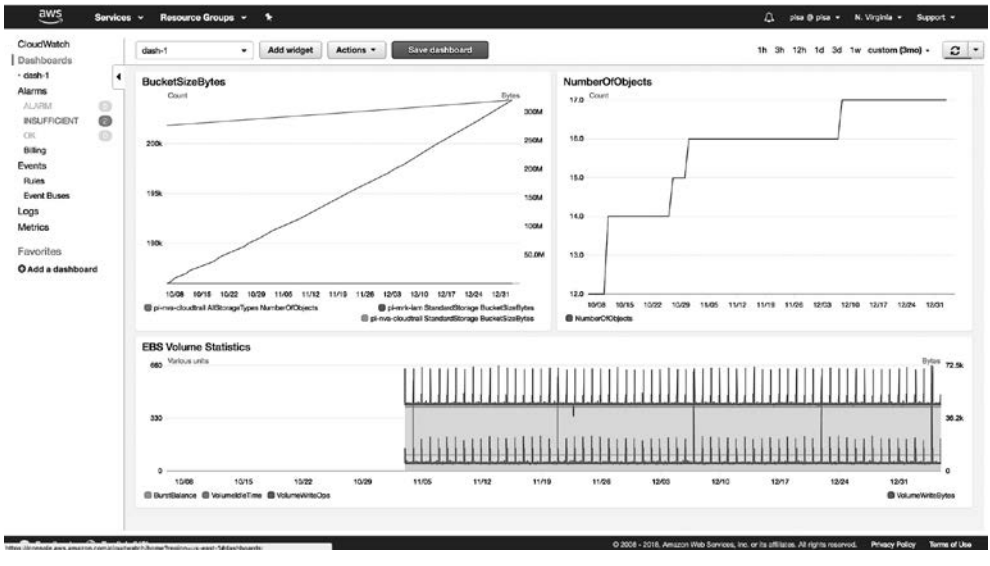


Рис. 5.9

Каждый график на скриншоте представляет собой настраиваемый виджет, который может отображать множество показателей, собранных из других сервисов, имеющихся на платформе. Пользовательские показатели также могут включаться в сервис и отслеживаться им.

Провайдеры CSP, отличные от AWS, предлагают аналогичные сервисы с постоянно расширяющимся набором функций и большими возможностями. Стоит отметить, что это характерная для данной индустрии тенденция — сторонние партнеры находятся под постоянным давлением со стороны провайдеров CSP, поскольку последние непрерывно увеличивают число своих услуг и функций.

Такой подход — благо для потребителей облачных услуг, поскольку в этой высококонкурентной среде вся отрасль быстрее развивается.

Практические рекомендации для архитектуры cloud native

Создайте несколько панелей мониторинга, каждая из которых даст вам представление об одном стеке приложений или соответствующей команде эксплуатации. Отношение «один к одному» позволит вам быстро и легко просматривать все релевантные данные для одного автономного стека, что значительно упростит устранение неполадок и настройку производительности. Этот подход можно использовать и для упрощения отчетности.

Опираясь на рис. 5.8, мы можем конкретизировать сервисы для разных случаев применения, как показано на схеме на рис. 5.10.

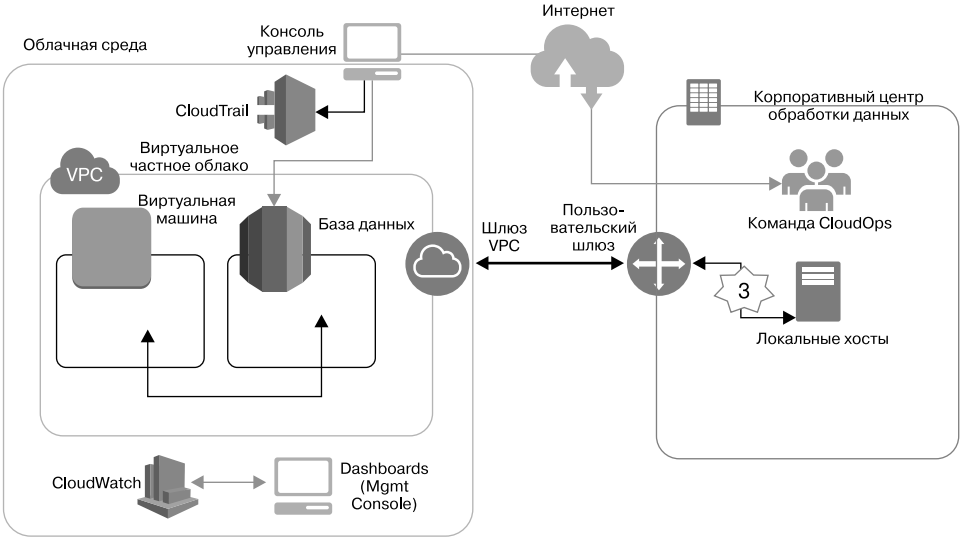


Рис. 5.10

AWS CloudTrail служит контрольным журналом для каждого вызова API, выполняемого в облачной среде. Это позволяет владельцам иметь полное представление о том, кто с какими ресурсами и какие действия выполняет. Amazon CloudWatch предоставляет встроенные возможности сбора показателей и создания отчетов. Эти пользовательские панели мониторинга могут собирать данные для составления отчетов даже из локальных хостов и систем.

Создание таких сервисов, как CloudWatch и CloudTrail, а затем формирование настраиваемой панели мониторинга на основе среды использования и поддерживаемого вами приложения — это первые важные шаги по настройке облачной среды и приложения.

Инфраструктура как код

Сама природа облака отделяет пользователя от ресурсов, которые ему выделяются и которые он использует. У него нет возможности физически устранять неполадки в неисправных или отказавших устройствах. Поэтому потребовалась оцифровка всей функциональности, которая ранее выполнялась путем перехода к ресурсам и взаимодействия с ними. Это, конечно, не единственная причина, но она была ведущим архитектурным направлением во времена зарождения облака. CSP создали свои платформы с нуля с помощью API. Любые сервисы и функции доступны для использования, конфигурирования, потребления или какого-то другого вида программного взаимодействия через набор определенных API. Синтаксис, описания, *интерфейсы командной строки (CLI)* и примеры применения публикуются и поддерживаются каждым CSP и доступны для свободного просмотра и использования всеми.

После того как определенная программная конфигурация была внедрена, стало возможно выразить всю ИТ-среду или стек через код. Так появилась концепция *инфраструктуры как кода (IaC)*. Многие из концепций, подходов и инструментов, которые разработчики задействовали для совместной работы, тестирования и развертывания своих приложений, теперь могут применяться к инфраструктуре (вычислительные ресурсы, базы данных, сети и т. д.). Достаточно сказать, что эта разработка стала одним из крупнейших вкладов в быстрое развитие ИТ-систем в целом, поскольку позволила архитекторам инфраструктуры итеративно развивать свои системы так же быстро, как и разработчикам.

Чтобы настроить легкодоступное, масштабируемое и отказоустойчивое приложение, облачным архитекторам важно принять многие принципы, лежащие сегодня в основе разработки программного обеспечения. Мы рассмотрим их подробнее в главе 8, но эволюционный подход, доступный при работе с IaC, — это фактор, способствующий достижению высокого уровня развития стека (с хорошими показателя доступности, масштабируемости и отказоустойчивости). Не думайте, что ваше облачное приложение будет зрелым и устойчивым с первого дня. Вы должны каждый день работать над повышением его эффективности и внедрять в него новые механизмы отказоустойчивости. Этот подход является залогом успеха.

Практические рекомендации для архитектуры cloud native

Управляйте набором стеков IaC вместо того, чтобы создавать один большой монолитный стек IaC. Обычный подход заключается в том, чтобы вынести конфигурацию и ресурсы базовой сети в один стек (обычно это одноразовая конфигурация, которая со временем претерпевает незначительные изменения и становится общим ресурсом, используемым во всех приложениях, поэтому требуется собственный выделенный стек). Выносите приложения в отдельные стеки или наборы стеков. Ресурсы, разделяемые приложениями, редко стоит включать в стек одного приложения — управляйте им в отдельном стеке, чтобы избежать конфликтов между двумя командами разработчиков.

Опираясь на рис. 5.6, мы можем определить всю инфраструктуру облачной сети как шаблон кода (назовем его сетевым стеком), как показано на рис. 5.11.

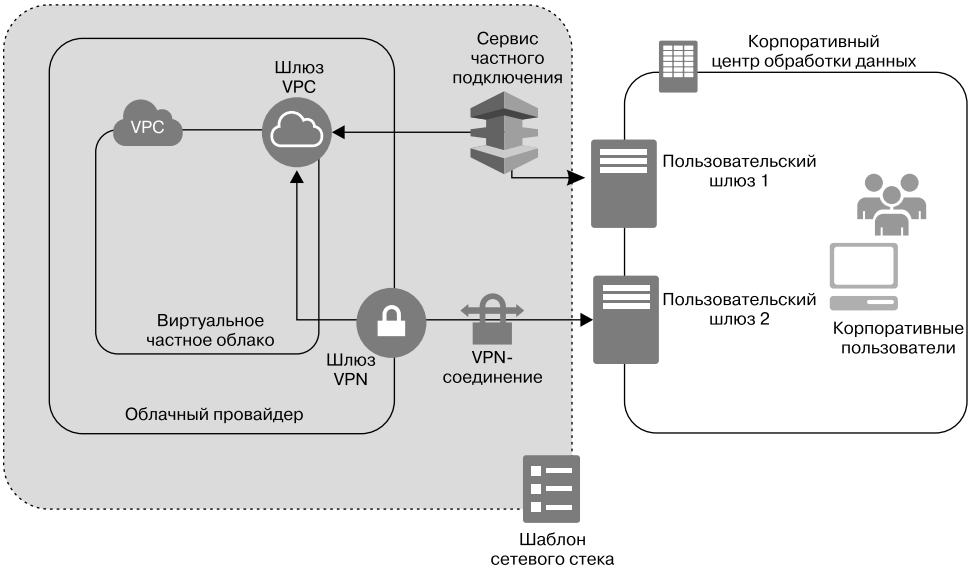


Рис. 5.11

Все, что находится на сером фоне, определяется как код в шаблоне сетевого стека. Любыми изменениями, которые необходимо внести в сетевой стек, можно управлять путем редактирования кода, содержащегося в этом стеке.

Основываясь на рис. 5.4, мы видим один подход к управлению приложением с помощью IaC. Конфигурацию веб-сервера, группу автоматического масштабирования и балансировщик гибкой нагрузки определяют в шаблоне веб-стека, там же ими и управляют. Конфигурацией базы данных и ресурсами управляют в шаблоне стека базы данных, как показано на рис. 5.12.

На этой схеме представлены сервисы cloud native и сторонние варианты. AWS CloudFormation, *Azure Resource Manager (ARM)* и GCP Deployment Manager — это примеры нативных сервисов CSP. Код может быть написан на языках *JavaScript Object Notation (JSON)* или *Yaml Ain't Markup Language (YAML)*, который ранее был известен как *Yet Another Markup Language*.

Популярный сторонний инструмент — Terraform от HashiCorp. Он работает во всех основных облаках и локально, что позволяет пользователям создавать единый ландшафт IaC в среде гибридной инфраструктуры. Terraform поддерживает собственный формат для конфигурационных файлов (их еще называют шаблонами), творчески названный форматом terraform, поддерживает он и формат JSON. Хотя это не совсем IaC, существует множество инструментов для управления конфигурацией, таких как Vagrant, Ansible, Chef и Puppet.

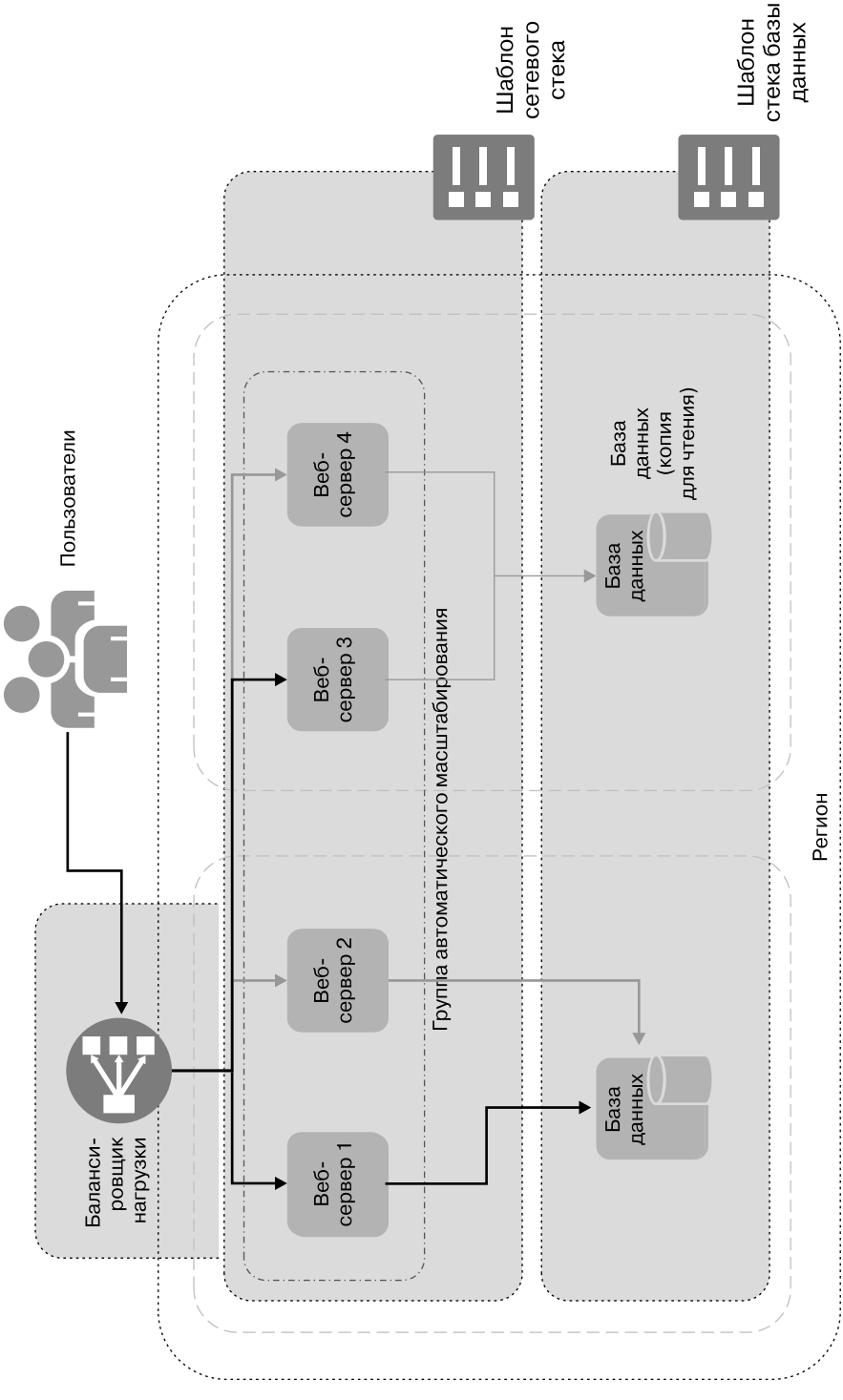

Шаблон сетевого стека
Шаблон стека базы данных
Пользователи
Веб-сервер 4
Веб-сервер 3
Веб-сервер 2
Веб-сервер 1
Группа автоматического масштабирования
База данных (копия для чтения)
База данных
База данных
Балансировщик нагрузки
Регион

Рис. 5.12

Для продвинутого развертывания в AWS, например, будет использоваться CloudFormation, а для автоматизации и уменьшения ошибок развертывания — инструмент управления конфигурацией, такой как Chef.

Неизменяемые развертывания

«Неизменяемый» означает, что что-то не изменяется со временем или не может быть изменено. Неизменяемое развертывание — это подход, при котором состояние стека приложения фиксируется после его развертывания и настройки. Если необходимы какие-либо изменения, весь стек приложения или элемент стека развертывается повторно.

Облачные виртуальные машины по своей природе не надежнее и долговечнее локальных машин. Их основное преимущество заключается в том, что они *стандартизированы* и *легко заменяемы*. Представьте, что вы строите замок из деталек Lego. Если вы сломаете один желтый кирпичик 4×2 см, вам не придется ждать его ремонта для продолжения стройки. Вы найдете такой же и продолжите строить. Подобное отношение уместно и для облачных ресурсов. Говоря более кратко и беспощадно, «относитесь к своим ресурсам как к скоту, а не как к домашним животным». Если виртуальная машина показывает плохую производительность, будьте готовы от нее избавиться, чтобы освободить ресурсы для ее исправных собратьев.

Если продолжить аналогию с Lego, будет полезно рассмотреть завершенную структуру, например замок, как неизменяемую. Любые изменения, которые мы захотим внести в замок (например, новый разводной мост), будут выполнены и протестированы во втором замке, который мы построим. Первый продолжит нормально функционировать, пока мы не завершим тестирование и не будем готовы пользоваться новым замком с разводным мостом.

Обрабатывая таким образом стеки приложений — каждый раз заменяя всю систему на минимально возможном уровне, — можно получить следующие преимущества.

- Замена системы на самом низком уровне побуждает вас автоматизировать каждый шаг развертывания.
- Возврат к предыдущей версии прост и понятен, поскольку исправный старый стек будет работать до тех пор, пока не протестируют и не запустят в работу новый стек. Сравните это с непрерывным обновлением уже развернутого стека. Если что-то пойдет не так, необходимо откатить обновления, надеясь, что получится отменить все сделанные изменения. Каждый этап непрерывного обновления и отката выводит систему из строя. Этого не случится при неизменяемом развертывании.
- Поскольку автоматизация желательна, каждое изменение должно быть прописано в виде сценария. Для команд, продолжающих вносить изменения вручную,

это будет непростым уроком, поскольку ни один сервер не является особенным и может быть в любой момент выведен из эксплуатации.

- Создание сценариев для всех случаев означает, что вы можете очень легко создавать производственные системы для разработки и тестирования, изменив несколько строк кода.
- Самое главное то, что новые инфраструктуры можно тестировать изолированно, не рискуя производственным стеком. В критически важных бизнес-приложениях это имеет первостепенное значение для защиты систем, приносящих доход.

Отношение к вашим развертываниям как к неизменяемым требует не просто обновления конфигурации, а настоящего сдвига в образе мышления и корпоративной культуре. Это стало реальным благодаря новым возможностям, которые предлагает облако, в частности IaC (будь то AWS CloudFormation, Azure ARM или Terraform).

Существуют и другие облачные инструменты, которые помогают обеспечить еще большую автоматизацию и облегчить бремя развертывания. Облачные сервисы, такие как AWS Elastic Beanstalk, берут на себя эксплуатационные аспекты развертывания, давая пользователю возможность сосредоточиться на разработке кода приложения и работе с ним. Человек просто загружает свой код в сервис, а Elastic Beanstalk делает остальное (некоторые параметры конфигурации задает пользователь). Такие сервисы, как Elastic Beanstalk, в основу развертывания кладут неизменяемость, что позволяет вам предоставлять новый стек 2.0 вместе с исходным стеком 1.0. Вы можете выполнять тестирование и постепенно принимать все больше трафика пользователя в версии 2.0. Переход происходит, когда вы уверены, что стек 2.0 готов к работе. Если новый стек неисправен, можно быстро вернуться к исходной среде. Это делается просто изменением настроек балансировки нагрузки для направления трафика из старого в новый стек или наоборот.

Кроме того, сейчас существуют облачные сервисы, которые обеспечивают жесткость конфигурации и помогают поддерживать неизменяемость. Такие сервисы, как AWS Config и Config Rules, помогают отслеживать изменения конфигурации с течением времени, позволяя пользователям следить за изменениями ресурсов и просматривать состояния. Эта информация может потребляться через API вашей собственной системой уведомлений/исправлений, или вы можете использовать правила конфигурации, чтобы автоматически выполнять определенные действия в зависимости от того, как изменились ресурсы.

Неизменяемость — ключевая парадигма проектирования, которую необходимо принять на пути к развитию облака. Управляя изменениями развертывания на самом низком уровне ресурсов, вы поощряете автоматизацию и повышаете доступность приложения.

Самовосстанавливающиеся инфраструктуры

Еще одна важная парадигма для облачных приложений, связанная с масштабируемостью и доступностью, — это самовосстанавливающиеся инфраструктуры. Самовосстанавливающаяся инфраструктура — это интеллектуальное развертывание, которое автоматически реагирует на известные и распространенные сбои. В зависимости от типа сбоя архитектура принимает соответствующие меры для исправления ошибки.

Самовосстановление может срабатывать на уровне приложений, системы и оборудования. Облако полностью взяло на себя ответственность за «аппаратное самовосстановление». Такого явления *технически* не существует, поскольку нам еще предстоит найти способ починить сломанные жесткие диски, перегруженные процессоры или заменить сгоревшую оперативную память без участия человека. Однако, с точки зрения потребителя облака, текущее положение дел имитирует идеалистическое будущее. У CSP есть сотрудники на местах, которые быстро чинят и заменяют отказавшую технику. Строго говоря, нам еще предстоит приблизиться к самовосстанавливающимся физическим инфраструктурам, поскольку вмешательство человека все еще необходимо. Однако, как потребителям облачных технологий, нам не нужно беспокоиться об этом, поскольку мы полностью изолированы от аппаратного уровня.

На уровне системы и приложений в нашем распоряжении есть множество методов, помогающих создавать самовосстанавливающиеся инфраструктуры и облачные приложения. Далее приведено несколько примеров. Кроме того, мы рассмотрим кое-какие инструменты, которые будут использоваться позже в этой главе.

- *Автомасштабируемые группы* (ASG, Auto-scaling groups) — прекрасный пример систем самовосстановления. Хотя мы обычно связываем ASG с масштабируемостью, их можно настроить так, чтобы заменять неисправные виртуальные машины новыми. Отправка пользовательских показателей работоспособности в вашу систему мониторинга является ключевым фактором применения этой возможности. Важно разрабатывать приложения как не имеющие состояния, так как это позволит передавать сеансы через одну виртуальную машину на другую в рамках ASG.
- *Проверки работоспособности DNS*, доступные на облачных платформах, позволяют пользователям следить за исправностью определенных ресурсов, состоянием встроенных сервисов мониторинга и работой других средств проверки работоспособности. Затем вы можете грамотно и автоматически перенаправить трафик с учетом работоспособности стека.
- *Автоматическое восстановление инстанса (виртуальной машины)* — это функция, предоставляемая CSP, например AWS, для автоматического восстановления неисправного инстанса в случае сбоя основного оборудования. Если нарушается сетевое подключение, пропадает питание системы, возникают проблемы

с программным обеспечением на физическом хосте или с оборудованием на физическом хосте, которые влияют на доступность сети, AWS реплицирует инстанс и уведомляет об этом пользователя.

- *Поддержка отказоустойчивости или кластеризации баз данных* доступна через управляемые сервисы БД у основных CSP. Такие сервисы, как AWS RDS (сервис реляционных баз данных), дают пользователям возможность предоставлять развертывания с технологией RDS Multi-AZ. Применяя зеркалирование SQL Server для SQL, развертывания Multi-AZ для Oracle, PostgreSQL, MySQL и MariaDB и кластеризацию для собственного движка баз данных Amazon под названием Aurora, эти сервисы поддерживают легкодоступные компоненты стека БД. В случае сбоя сервиса база данных автоматически переключается на синхронизированную резервную копию.

Основные принципы

Приложения cloud native должны быть масштабируемыми и доступными. Чтобы достичь этого, отказоустойчивые системы в облаке должны поддерживать концепции, описанные в данной главе. Мы суммировали основные принципы, которые необходимо соблюдать для создания облачных приложений, следующим образом.

- *Вычисления должны быть распределены, приложение — не иметь состояния.*
  - Используйте распределение по нескольким зонам и географическим регионам. Воспользуйтесь всеми преимуществами гиперпоставщиков облачных решений и создайте собственные системы, которые будут мультизональными или мультирегиональными.
  - Приложение с сохранением состояния хранит данные о каждом сеансе на машине и использует их, пока сеанс активен. Однако приложение без сохранения состояния не сохраняет никаких данных о состоянии сеанса на сервере/хосте. Вместо этого данные сеанса хранятся на клиенте и передаются на сервер по мере необходимости. Это позволяет взаимозаменяемо применять вычислительные ресурсы (например, когда ASG расширяют или сокращают количество вычислительных ресурсов в группе).
- *Не храните свои данные локально и распределяйте их.*
  - Используйте облачные сервисы хранения, которые обеспечивают избыточность ваших данных в соответствии с дизайном. Сервисы AWS, такие как *Simple Storage Service (S3)* и *Elastic Block Store (EBS)*, автоматически реплицируют данные на несколько зон доступности и несколько жестких дисков.
  - Применение решений распределенного кэширования, таких как Redis и Memcache, помогает поддерживать архитектуры приложений без сохранения состояния.

  - Используйте в качестве хранилищ информации решения с возможностью распределения данных и устойчивостью к сбоям. Развертывание баз данных с несколькими зонами доступности или использование полностью управляемых сервисов NoSQL DB, таких как AWS DynamoDB, обеспечивают избыточность и отказоустойчивость.

- *Создайте физически избыточные сетевые подключения.* Даже самые крутые приложения могут выйти из строя, если сетевое подключение разорвано и отсутствует план аварийного переключения. Продумайте резервное сетевое подключение к облаку из существующего центра обработки данных или точки обмена сетевым трафиком. Это основополагающее решение при построении сети, которое принесет плоды в случае экстремального сбоя. Этот подход распространяется и на сетевое оборудование, размещенное локально. Частный канал и его резервное VPN-соединение не должны использовать одни и те же сетевые устройства. Имейте хотя бы два физически разделенных ресурса с высокой надежностью.
- *Выполняйте тщательный и всеобъемлющий мониторинг.* Без продуманного, тщательного и непрерывного мониторинга вы не сможете создать масштабируемую и легкодоступную систему. Выполняя мониторинг на всех уровнях стека, вы обеспечите автоматическое включение самовосстанавливающихся инфраструктур и сможете выявлять режимы работы, в которых часто возникают сбои. Кроме того, по мере накопления данных о поведении системы выявляются закономерности, с учетом которых можно интеллектуально масштабировать трафик и обходить узкие места в производительности системы.
- *Используйте IaC.* Выражая стек как код, вы можете использовать все преимущества методологий гибкой разработки программного обеспечения для развертывания ИТ-систем и управления ими. Это очень важный момент для создания облачных приложений, поскольку без него некоторые концепции, представленные в этой главе, были бы невозможны.
- *Каждое развертывание должно быть неизменяемым.* Устанавливайте политику принятия изменений в системе на нижнем уровне. Если вам необходимо обновить приложение, внести в него исправление, разверните новую виртуальную машину на основе обновленного образа или запустите новый контейнер с обновленным приложением. Это существенно изменяет не только технологический, но и культурный уклад организации.
- *Проектируйте и внедряйте инфраструктуру самовосстановления, когда это возможно.* Сократите эксплуатационные расходы, развертывая ресурсы и создавая системы, которые восстанавливаются самостоятельно. Будьте готовы к сбоям даже в облаке. Сервисы CSP и виртуальные машины не защищены от сбоев оборудования или сети, и вы не должны ожидать стопроцентно безотказной работы. Примите это во внимание и проведите распределенное развертывание в нескольких географических зонах.
- *Автоматически регистрируйте и предотвращайте ошибки в развертывании.* Благодаря обработке IaC и принятию неизменяемых развертываний каждое

изменение в стеке можно отслеживать и проверять. Встраивайте проверки конфигурации и правила в среду, чтобы предотвратить изменения. Внедрите проверки кода в свой конвейер, чтобы пометить несовместимые системы перед их развертыванием. Это будет более подробно описано в главе 8.

- *Создайте панель оперативного управления.* Создавайте пользовательские панели, адаптированные к вашему приложению и системе. Задействуйте комбинацию встроенных сервисов ведения журнала и пользовательских метрик, чтобы выводить все критически важные данные на одном экране. Кроме того, создайте правила для уведомления администраторов, когда показатели превышают нормальные значения.
- *Масштабируемость против гибкости — знайте разницу и разрабатывайте архитектуру соответствующим образом.* Масштабируемость, обеспечиваемая облаком, дает пользователям практически неограниченную возможность расширять свои приложения. Требования к масштабируемости приложения будут зависеть от того, где находятся ваши пользователи, каковы ваши глобальные амбиции и какой конкретный тип ресурса необходим. Гибкость — это архитектурная особенность, которую вы должны продумать и встроить в свою систему. Это означает, что система может расширяться и сокращаться в соответствии с требованиями пользователя. Это оптимизирует производительность и затраты.

Сервис-ориентированные архитектуры и микросервисы

Истоки развития облачных вычислений для ведущих CSP связаны с трудностями управления монолитными системами. В то время как компания (сайт розничной торговли Amazon.com, поисковая система Google.com) переживала колоссальный рост, традиционные ИТ-системы не успевали за темпами развития и не обеспечивали внедрения необходимых инноваций.

Это правдивая история, относящаяся как к AWS, так и к GCP, и ее подробное описание является общедоступным. По мере того как вокруг облачных сервисов компании (изначально предоставляемых как внутренние исключительно для групп разработчиков продуктов) формировались их внутренние системы, компания Amazon сделала шаг по предоставлению внешним клиентам этих сервисов, основанных на биллинговой системе.

Многие компании и архитекторы теперь видят преимущества процесса разделения, называя полностью отделенную среду *сервис-ориентированной архитектурой (service-oriented architecture, SOA)*. Мы определяем SOA как *цифровую среду, в которой каждая составляющая системы, обеспечивающая самый низкий уровень функциональности приложения, запускается независимо и взаимодействует с другими системами исключительно через API.* На практике это означает, что каждый

сервис развертывается и управляется независимо, в идеале как стек, управляемый IaC. Сервисы могут зависеть от других сервисов, однако этот обмен данными осуществляется через API.

Инструменты для развертывания в облаке

Теперь, когда у нас есть четкое понимание и стратегии создания масштабируемых и доступных систем, мы можем ознакомиться с инструментами, продуктами и проектами с открытым исходным кодом, помогающими реализовать эти стратегии. Эти инструменты помогут вам разработать более продвинутые системы по направлениям автоматизации и прикладного проектирования CNMM.

Simian Army

Simian Army — это набор инструментов облачного тестирования, созданный компанией Netflix и размещенный на GitHub в качестве проекта с открытым исходным кодом. Эта «армия» состоит из «обезьян» (различных инструментов) от Chaos Monkey и Janitor Monkey до Conformity Monkey. Каждая «обезьяна» специализируется на определенной задаче, такой как отключение подсистемы, удаление недостаточно активно работающих ресурсов или ресурсов, которые не соответствуют предопределенным правилам. Использование инструмента Simian Army — отличный способ протестировать ваши системы в *игровой ситуации*, когда инженеры и администраторы находятся на местах и готовы починить систему, если она не справится с результатами атаки «обезьяньей» армии. Netflix запускает работу этих инструментов только в течение рабочего дня, что позволяет инженерам постоянно проверять эффективность своей архитектуры. Поскольку армия рушит подсистемы, самым суровым испытаниям подвергаются механизмы самовосстановления приложений.

Инструмент Simian Army доступен на GitHub по адресу https://github.com/Netflix/SimianArmy.

Docker

Docker — это программная контейнерная платформа, которая позволяет упаковать приложение со всеми его зависимостями и необходимыми библиотеками в самодостаточный модуль. Она позволяет организовать архитектуру системы вокруг контейнерных приложений вместо виртуальных машин. Контейнеры помогают устранить риск несовместимости или конфликта версий между приложением и базовой ОС. Кроме того, их легко перенести на любую машину с Docker.

Поскольку это относится к облачным приложениям, контейнеры являются эффективным средством упаковки приложений во время выполнения, отлично подходят для управления версиями и эффективной эксплуатации развертываний. Docker — ведущее решение в этой области, и его вариант *Community Edition (CE)* можно свободно использовать.

В дополнение к Docker есть несколько заслуживающих внимания проектов с открытым исходным кодом, которые могут помочь максимально эффективно применять Docker и создавать облачные системы.

- *Infrakit* — это инструментарий для оркестрации инфраструктуры. Он оставляет неизменяемым подход к архитектуре и делит процессы автоматизации и администрирования инфраструктуры на простые компоненты. Infrakit обеспечивает инфраструктурную поддержку высокоуровневых систем управления контейнером и делает вашу инфраструктуру самовосстанавливающейся. Он имеет множество плагинов для других популярных инструментов и проектов, таких как Swarm, Kubernetes, Terraform и Vagrant, а также поддерживает среды AWS, Google и VMWare (https://github.com/docker/deploykit).
- *Docker Swarm(kit)* — инструмент управления кластером и оркестрации для Docker Engine 1.12 и более поздних версий. Он позволяет вашему развертыванию контейнера Docker поддерживать согласование сервисов, балансировку нагрузки, обнаружение сервисов и ротацию встроенных сертификатов. Доступна также автономная версия Docker Swarm, но ее развитие прекратилось, когда начала расти популярность Swarmkit (https://github.com/docker/swarmkit).
- *Portainer* — это легкий интерфейс управления для сред Docker. Он прост в использовании и работает в одном контейнере, который может действовать на любом движке Docker. При создании эксплуатационной панели управления такие интерфейсы, как Portainer, необходимы, чтобы составить общее представление об окружающей среде и системах. (Перейдите на https://github.com/portainer/portainer или https://portainer.io/, чтобы узнать больше).

Kubernetes

С ростом популярности и успеха Docker в середине 2010-х компания Google признала тенденцию к контейнеризации и внесла значительный вклад в ее внедрение. Взяв за основу свою внутреннюю систему оркестрации инфраструктуры системы под названием Borg, компания Google с нуля разработала Kubernetes, чтобы организовать контейнеры в Docker. Kubernetes — это мощный инструмент с открытым исходным кодом, который наряду с обеспечением многих других возможностей помогает монтировать системы хранения, проверять работоспособность приложений, реплицировать экземпляры приложений, автоматически масштабировать, развертывать обновления, балансировать нагрузку, мониторить ресурсы и отлаживать приложения. В 2016 году Kubernetes был передан в дар

Cloud Native Compute Foundation, что обеспечило его бесплатное использование и поддержку активными членами сообщества (https://kubernetes.io/).

Terraform

Terraform от Hashicorp — это популярный инструмент для создания версий *IaC*, их изменения и управления ими. В отличие от собственных облачных сервисов, таких как AWS CloudFormation или Azure ARM, Terraform работает с CSP и локальными центрами обработки данных. Это дает явное преимущество при обработке ресурсов инфраструктуры в гибридной среде (https://github.com/hashicorp/terraform).

OpenFaaS (функция как сервис)

OpenFaaS, сочетающий в себе несколько инструментов, упомянутых ранее (Docker, Docker Swarm и Kubernetes), — это проект с открытым исходным кодом, позволяющий легко создавать бессерверные функции. Он помогает автоматизировать развертывание шлюза API для вызова ваших функций и доступа к собранным метрикам. Шлюз масштабируется в соответствии с потребностями, а пользовательский интерфейс доступен для мониторинга (https://github.com/openfaas/faas).

Envoy

Первоначально разработанный в Lyft, Envoy — это пограничный и сервисный прокси-сервер с открытым исходным кодом, созданный для облачных приложений. Прокси-сервисы являются ключевым элементом внедрения сервис-ориентированных архитектур, расширенные реализации которых обеспечивают потрясающие масштабируемость и доступность (https://www.envoyproxy.io/).

Linkerd

Linkerd — еще один проект с открытым исходным кодом, поддерживаемый Cloud Native Compute Foundation. Это популярный проект сервисной сетки (service mesh). Он разработан для того, чтобы сделать архитектуры SOA масштабируемыми за счет прозрачного внедрения механизмов обнаружения сервисов, балансировки нагрузки, обработки отказов и интеллектуальной маршрутизации во все аспекты межсервисного взаимодействия (https://linkerd.io/).

Zipkin

Zipkin — это проект с открытым исходным кодом, реализующий на практике идеи из научной работы под названием «Dapper, крупномасштабная инфраструктура

отслеживания распределенных систем», опубликованной подразделением Google Research. Zipkin помогает архитекторам устранять неполадки и оптимизировать задержку в своих микросервисных архитектурах, асинхронно собирая данные на определенном временном интервале и упорядочивая их с помощью пользовательского интерфейса (https://zipkin.io/).

Ansible

Ansible — популярная платформа управления конфигурациями и развертыванием с открытым исходным кодом, являющаяся частью программного обеспечения RedHat. Она основана на безагентной связи между рабочими и управляющими узлами. Данная платформа поддерживает схемы автоматизации и развертывания стеков инфраструктуры приложений (https://www.ansible.com/).

Apache Mesos

По сути, Mesos — это менеджер кластеров, который дает вашим приложениям возможность масштабироваться за пределы одной виртуальной машины или физического компьютера. Mesos позволяет легко создавать распределенную систему по пулу разделяемых ресурсов. Предоставляя общие функциональные возможности, в которых нуждается любая распределенная система (например, обнаружение сбоев, распределение, запуск, мониторинг, удаление и очистка задач), программа Mesos стала популярным выбором для поддержки развертывания структуры Hadoop для больших данных. Приложение Mesos было создано в Калифорнийском университете в AMPLab в Беркли и стало проектом верхнего уровня Apache Foundation в 2013 году (http://mesos.apache.org/).

Saltstack

Другой проект по управлению конфигурацией с открытым исходным кодом и проект оркестрации — Saltstack — является опцией для управления облачным развертыванием в масштабе. Подумайте об этом приложении, если в вашем проекте преимущественно используются дистрибутивы ОС на базе Linux. Приложение Salt было написано Томасом С. Хетчем и выпущено в 2011 году (https://github.com/saltstack/salt).

Vagrant

Vagrant — проект с открытым исходным кодом, принадлежащий Hashicorp. Он предоставляет согласованный метод настройки виртуальных сред. При использовании в сочетании с ПО автоматизации Vagrant обеспечивает масштабируемость

и переносимость во всех виртуальных средах. Первый выпуск Vagrant был написан в 2010 году Митчеллом Хасимото и Джоном Бендером (https://www.vagrantup.com/).

Проекты OpenStack

OpenStack — комплекс проектов свободного программного обеспечения, который может быть использован для создания инфраструктурных облачных сервисов и облачных хранилищ с открытым исходным кодом, имитирующих все сервисы, имеющиеся в современных ведущих CSP. Проекты OpenStack имеют широкий охват — от вычислений, хранения данных и организации сетей до сбора и анализа информации, обеспечения безопасности и реализации прикладных сервисов.

Вычисления:

- Nova — контроллер вычислительных ресурсов, https://wiki.openstack.org/wiki/Nova;
- Glance — библиотека образов виртуальных машин, https://wiki.openstack.org/wiki/Glance;
- Ironic — подготовка OpenStack на чистом железе, https://wiki.openstack.org/wiki/Ironic;
- Magnum — обеспечение движка оркестрации контейнеров, https://wiki.openstack.org/wiki/Magnum;
- Storlets — хранилище вычисляемых объектов, https://wiki.openstack.org/wiki/Storlets;
- Zun — сервис контейнеров, https://wiki.openstack.org/wiki/Zun.

Хранение, резервное копирование и восстановление:

- Swift — облачное файловое хранилище, https://wiki.openstack.org/wiki/Swift;
- Cinder — сервис работы с блочными устройствами хранения данных, https://wiki.openstack.org/wiki/Cinder;
- Manila — общие файловые системы, https://wiki.openstack.org/wiki/Manila;
- Karbor — проекция данных приложения как сервиса, https://wiki.openstack.org/wiki/Karbor;
- Freezer — резервное копирование, восстановление и аварийное восстановление, https://wiki.openstack.org/wiki/Freezer.

Сеть и доставка контента:

- Neutron — сеть, https://docs.openstack.org/neutron/latest/;
- Designate — сервис DNS, https://wiki.openstack.org/wiki/Designate;
- DragonFlow — плагин к Neutron, https://wiki.openstack.org/wiki/Dragonflow;
- Kuryr — контейнерный плагин, https://wiki.openstack.org/wiki/Kuryr;

- Octavia — балансировщик нагрузки, https://wiki.openstack.org/wiki/Octavia;
- Tacker — оркестрация NFV, https://wiki.openstack.org/wiki/Tacker;
- Tricircle — сетевая автоматизация для развертываний в нескольких регионах, https://wiki.openstack.org/wiki/Tricircle.

Данные и аналитика:

- Trove — база данных как сервис, https://wiki.openstack.org/wiki/Trove;
- Sahara — предоставление инфраструктуры обработки больших данных, https://wiki.openstack.org/wiki/Sahara;
- Searchlight — индексирование и поиск, https://wiki.openstack.org/wiki/Searchlight.

Безопасность, идентификация и соответствие требованиям:

- Keystone — сервис идентификации, https://wiki.openstack.org/wiki/Keystone;
- Barbican — управление ключами, https://wiki.openstack.org/wiki/Barbican;
- Congress — управление, https://wiki.openstack.org/wiki/Congress;
- Mistral — сервис рабочего пространства, https://wiki.openstack.org/wiki/Mistral.

Инструменты управления:

- Horizon — графический интерфейс администрирования, https://wiki.openstack.org/wiki/Horizon;
- Openstack Client — клиент командной строки, https://www.openstack.org/software/releases/ocata/components/openstack-client-(cli);
- Rally — сервис тестирования производительности, https://rally.readthedocs.io/en/latest/;
- Senlin — сервис кластеризации, https://wiki.openstack.org/wiki/Senlin;
- Vitrage — сервис анализа первопричин, https://wiki.openstack.org/wiki/Vitrage;
- Watcher — сервис оптимизации ресурсов, https://wiki.openstack.org/wiki/Watcher.

Инструменты развертывания:

- Chef Openstack — рецепты автоматизации для OpenStack, https://wiki.openstack.org/wiki/Chef;
- Kolla — развертывание контейнеров, https://wiki.openstack.org/wiki/Kolla;
- OpenStack Charms — надстройка над Juju для OpenStack, https://docs.openstack.org/charm-guide/latest/;
- OpenStack-Ansible — управление задачами Ansible для OpenStack, https://wiki.openstack.org/wiki/OpenStackAnsible;

- Puppet OpenStack — ИТ-автоматизация для облачных развертываний OpenStack, https://docs.openstack.org/puppet-openstack-guide/latest/;
- Tripleo — сервис развертывания, https://wiki.openstack.org/wiki/TripleO.

Прикладные сервисы:

- Heat — оркестратор, https://wiki.openstack.org/wiki/Heat;
- Zaqar — сервис обмена сообщениями, https://wiki.openstack.org/wiki/Zaqar;
- Murano — каталог приложений, https://wiki.openstack.org/wiki/Murano;
- Solum — автоматизация жизненного цикла разработки программного обеспечения, https://wiki.openstack.org/wiki/Solum.

Мониторинг и учет:

- Ceilometer — сервис учета и сбора данных, https://wiki.openstack.org/wiki/Telemetry;
- Cloudkitty — биллинг и возвратные платежи, https://wiki.openstack.org/wiki/CloudKitty;
- Monasca — мониторинг, https://wiki.openstack.org/wiki/Monasca;
- AODH — сервис оповещений, https://docs.openstack.org/aodh/latest/;
- Panko — сервис индексации событий, метаданных, https://docs.openstack.org/panko/latest/.

Существует еще множество доступных инструментов (с открытым исходным кодом и нет), и многие готовящиеся проекты находятся в стадии разработки. Данный список инструментов не следует воспринимать как окончательный — это просто краткий перечень некоторых наиболее популярных инструментов, используемых на момент написания этой книги. Важный вывод: автоматизация и свобода — это ключевые факторы. Подробнее о выборе и использовании инструментов для эксплуатационной работы в облаке мы поговорим в главе 8.

Резюме

В этой главе мы познакомили вас с использованием облаков в самых крупных масштабах и всеми последствиями объединения организованных вычислительных средств для потребителей ИТ. Глобальный масштаб, согласованность и охват этих облачных платформ изменили наше представление о масштабируемых и доступных системах.

Мы представили концепцию постоянно работающих архитектур и ключевые архитектурные элементы, составляющие эти системы. Резервирование сетевых

ресурсов и основных сервисов, расширенный мониторинг, IaC и неизменяемые развертывания — все это важные элементы построения любой облачной системы.

Опираясь на этот постоянный подход, мы ввели понятие самовосстанавливающейся инфраструктуры. Для крупномасштабных облачных развертываний автоматизация восстановления системы — ключевая функция. Это дает возможность системам восстанавливаться самостоятельно, но, что более важно, высвобождает критически важное время специалистов для работы над улучшением системы, что позволяет развивать архитектуры эволюционным путем.

Мы завершили эту главу перечнем некоторых самых популярных инструментов, доступных сегодня архитекторам и ИТ-специалистам. Эти инструменты используются для управления конфигурациями, автоматизации, мониторинга, тестирования и управления сетями микросервисов.

В следующей главе поговорим о безопасности в облачных архитектурах.

6 Безопасность и надежность

Принимая решение о внедрении новой технологии на предприятии, большинство руководителей в первую очередь думают о том, как это отразится на безопасности. Возможность безопасного развертывания, а в дальнейшем обеспечения безопасности и реагирования на угрозы имеет первостепенное значение для успеха. Так было на заре появления компьютерных систем, и так будет в обозримом будущем. Из-за раскрытия или утечки данных клиентов, а также неправильного управления ими снижается общая безопасность ИТ-системы вплоть до вероятности потери бизнеса. Только за последнее десятилетие появились десятки примеров того, как предприятия разоряются из-за инцидентов с безопасностью.

Представители одной из крупнейших в прошлом поисковых систем Yahoo! во время переговоров с Verizon о покупке объявили, что компания стала жертвой атаки. В 2014 году хакеры получили доступ к настоящим именам, адресам электронной почты, датам рождения и номерам телефонов 500 млн пользователей. Несколько месяцев спустя было опубликовано новое объявление Yahoo! о нарушении безопасности, в результате которого был скомпрометирован 1 млрд учетных записей и паролей. Эти события снизили стоимость Yahoo! для компании Verizon примерно на 350 млн долларов.

Мы знаем десятки примеров, когда плохое управление, неадекватная реакция и неправильные архитектурные решения привели к одинаково плачевным результатам для компаний. В 2014 году были скомпрометированы 145 млн учетных записей пользователей eBay, в Heartland Payment Systems в 2008 году произошла утечка данных о 134 млн кредитных карт пользователей, у Target в 2013 году были украдены сведения о 110 млн кредитных/дебетовых карт и контактные данные клиентов, и этот список можно продолжить. Каждое нарушение безопасности обходится этим компаниям в миллионные штрафы, подрыв доверия потребителей и неисчислимые потери доходов от бизнеса.

Мы знаем, что безопасность — важный элемент любой ИТ-системы, так почему же и как эти компании потерпели неудачу? Ведь они определенно не испытывали недостатка в ресурсах или сотрудниках. Ответ на этот вопрос приводит нас к технологиям обеспечения безопасности в облаке.

До того как появились облачные технологии, ИТ-активы размещались в централизованно управляемом локальном хранилище. В лучшем случае ограничивался физический доступ к центрам обработки данных и за ними велось наблюдение (часто контроль был слабым). В худшем случае вычислительные ресурсы были рассредоточены по нескольким точкам, что не позволяло отслеживать и контролировать физические активы. Рассогласованное оборудование, плохой контроль доступа и несовершенное управление накладными расходами могут снизить уровень безопасности.

Общий подход к безопасности должен заключаться в изолировании этих ресурсов жесткой внешней оболочкой. Ее защита зачастую равносильна защите всего ИТ-ландшафта. Такой подход позволяет игнорировать угрозы, возникающие изнутри или проникающие через оболочку. Эта оболочка часто представляет собой брандмауэр, размещенный на критическом сетевом узле, где можно отслеживать весь трафик. IDS/IPS (системы обнаружения вторжений и системы защиты от вторжений соответственно) развернуты аналогичным образом. Когда угроза может обойти защищенную точку сетевого обмена, весь стек, работающий за брандмауэром, становится целью атаки. Существует совсем немного средств, способных помешать злоумышленнику, оказавшемуся внутри защитной оболочки, использовать уязвимости. Кроме того, очень мало вариантов защиты от угроз, исходящих изнутри этой оболочки (рис. 6.1).

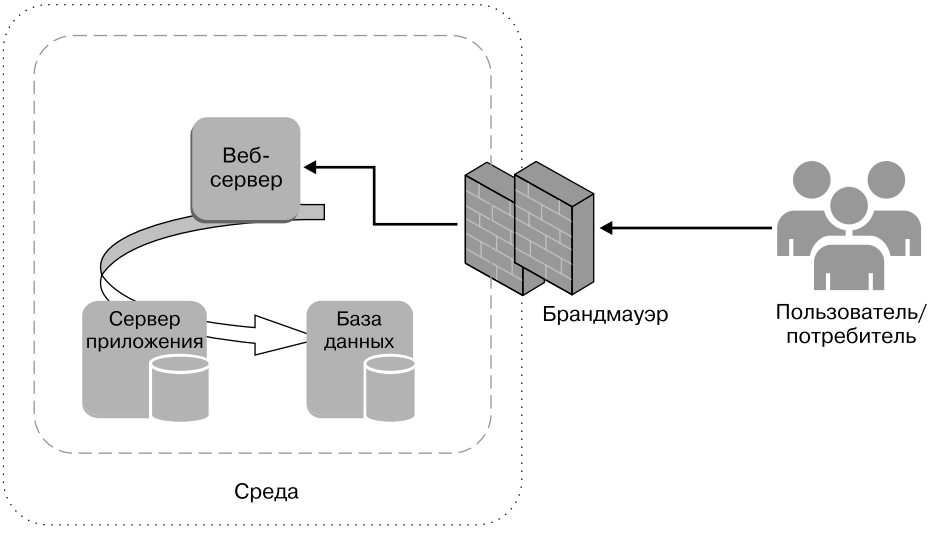


Рис. 6.1

Этот подход к безопасности, а также процедуры и организационная модель, которые зачастую идут с ним рука об руку, являются устаревшими. Команды безопасности,

группы разработки или инфраструктуры подотчетны разным менеджерам и преследуют различные цели. Группа, занимающаяся безопасностью, обычно не ориентирована на конечные бизнес-цели (захват большей доли рынка, увеличение доходов, снижение затрат на инфраструктуру и т. д.), поэтому ее действия воспринимаются как препятствующие развитию и инновациям.

Часто такое восприятие недалеко от реальности, но не по вине инженеров по безопасности. Сама организационная модель работы инженеров по безопасности обусловливает споры с разработчиками и архитекторами инфраструктуры. Вместо того чтобы быть партнерами, работающими над достижением одной и той же цели, они относятся друг к другу враждебно, это становится нормой, из-за чего результат оказывается нулевым. Для архитекторов и разработчиков стало обычным делом сетовать на то, что инженеры по безопасности делают процесс негибким. С одной стороны мы имеем разработчиков, стремящихся внедрять инновации и исправлять ошибки как можно раньше, а с другой — отдел безопасности, ориентированный на контроль и перепроверку вносимых изменений.

Безопасность в облачном мире

Что, если бы мы могли реорганизовать и переориентировать наших специалистов и сделать безопасность *средством* разработки, а не препятствием? Есть несколько способов преобразовать общедоступные облачные платформы.

- Предоставление и расширение доступа платформы к продуктам/услугам безопасности стандартными средствами.
- Предоставление встроенных функций безопасности в виде API.
- Интеграция функций безопасности в основные ИТ-сервисы (сеть, управление идентификацией и доступом, шифрование, DLP и др.).
- Кроме того, у общедоступных облачных платформ имеются некоторые изначальные преимущества.
    - Модель совместной ответственности, в которой обязанности по обеспечению безопасности берут на себя облачный провайдер и группа безопасности клиента.
    - Широкомасштабные и глубокие меры безопасности, предоставляемые облачным провайдером.
    - Обновления и функции, включенные в сервисы на постоянной основе.

У каждого гипероблачного провайдера есть команды профессионалов мирового уровня, насчитывающие тысячи человек. Их ежедневная работа состоит в том,

чтобы бороться с бесчисленным множеством существующих угроз и позволить потребителям облака успешно избегать этих проблем. Облачные провайдеры постоянно совершенствуют предоставляемые сервисы. Начнем с *управления идентификацией и доступом (identity and access management, IAM)*, *брандмауэров* и *брандмауэра веб-приложений (web application firewall, WAF)*, а затем перейдем к системам *агентской безопасности ОС*, такой как Amazon Inspector или Azure Security Center, и *предотвращения потери данных (data loss prevention, DLP)*, например Amazon Macie.

Безопасность на всех уровнях

Центрам обработки данных или облачным средам доступно множество вариантов брандмауэров и устройств VPN. Когда дело доходит до их выбора, важными факторами становятся не только безопасность, но и развертывание, а также доступные функции, включая средства управления доступом, возможность аутентификации и авторизации. Важно встроить эти функции на уровне приложения, вместо того чтобы полагаться исключительно на сетевую безопасность. Реализация безопасности на уровне приложений делает управление доступом масштабируемым, портативным и неизменяемым. Кроме того, доступ может регулироваться на основе реальной идентичности приложения или сервиса (микросервиса), а не на основе предоставления человеком.

По нашему определению, облачное приложение — это защищенное приложение. Логически рассуждая, защищенная система — это надежная система. Код приложения упаковывается и развертывается в нескольких облачных регионах, выполняется в различных контейнерах, и к нему обращаются многие клиенты или другие приложения (подробности см. в главе 5). Это делает включение безопасности в каждый микросервис еще более важным. Тот простой факт, что приложения теперь рассредоточены по нескольким географическим областям, к тому же постоянно увеличиваются и уменьшаются в объеме, требует более автоматизированного и детального подхода к обеспечению безопасности.

На протяжении большей части современной корпоративной истории написание кода безопасности было менее интересным и важным, чем создание бизнес-логики. Часто задачи, связанные с безопасностью, в цикле разработки решаются в последнюю очередь, из-за чего приходится идти на серьезные компромиссы при написании функций безопасности продукта. Если вы пытаетесь создать собственное облачное решение, которое может масштабироваться для поддержки больших объемов, работать в разных регионах и самообслуживаться, тогда главный архитектор обязательно рассматривает безопасность как один из основных строительных блоков архитектуры.

Если безопасность должна встраиваться в проект с нуля, а не привязываться к решению в последнюю минуту, то следует реализовать несколько важных функций безопасного облачного стека. Рассмотрим их.

- *Соответствие*. Обеспечение соответствия системы нормативно-правовым требованиям и ведение журнала аудита с помощью автоматизации. Создавайте журналы изменений, которые можно использовать для отчетов о соответствии (например, GxP, FFIEC и т. д.). Разверните правила, чтобы предотвратить несанкционированные методы развертывания.
- *Шифрование*. Конфиденциальные данные должны быть зашифрованы при передаче по сети или хранении в целевых хранилищах. Такие протоколы, как IPSec и SSL/TLS, необходимы для защиты потоков данных, проходящих через несколько сетей.
- *Масштабируемые и доступные ресурсы шифрования*. Не полагайтесь на один ресурс для выполнения функций шифрования. Эти ресурсы, как и любые другие облачные сервисы, должны быть рассредоточены и децентрализованы. Это не только обеспечивает лучшую производительность, но и устраняет единую точку отказа.
- *DLP*. Не позволяйте записывать какие-либо личные или конфиденциальные данные в журнал или другие неавторизованные места. Журналы довольно небезопасны и часто содержат информацию в текстовом виде. Потоки журналов зачастую становятся самой легкой целью для злоумышленников.
- *Защитите учетные данные и конечные точки*. Не храните учетные даные сервисов и исходные/целевые конечные точки в памяти. Используйте собственные сервисы токенизации с минимальными привилегиями, чтобы свести к минимуму масштаб потенциального ущерба. При создании учетных данных для сотрудников-операторов разработайте правила управления идентификацией и доступом (IAM) с отдельными и конкретными политиками. Постоянно просматривайте и отслеживайте их применение.
- *Кэширование*. Облачные приложения могут масштабироваться в нескольких экземплярах. Приложения должны использовать внешний кэш (например, Memcache и Redis) для поддержки проекта без сохранения состояния. Приложения никогда не должны хранить информацию в памяти дольше, чем необходимо для выполнения запроса. Следовательно, в случае сбоя одной машины запрос может быть легко перенесен на другую машину в парке серверов приложений.

Сервисы облачной безопасности

Как обсуждалось в предыдущих главах, одним из преимуществ разработки в облачной среде является получаемый разработчиками облачных технологий доступ к десяткам сервисов, которые легко интегрируются в среду. Это особенно важно,

когда речь идет о безопасности, поскольку в платформу встроено множество функций, которые пользователь может и должен применять. Далее приводится список действующих сервисов и функций.

Сетевые брандмауэры

Группы безопасности (SG) и списки контроля доступа к сети (NACL) действуют как брандмауэры для виртуальных машин в плоскости вашей облачной сети. SG действуют через *интерфейс машинной сети (network interface, NI)* и, как правило, более гибки и полезны при повседневном развертывании. SG могут быть изменены на лету, а правила распространяются на все NI в группе. SG по умолчанию ограничивают весь входящий трафик, кроме трафика от других машин в той же SG, и разрешают весь исходящий трафик. NACL похожи, но применяются ко всей подсети и по умолчанию разрешают весь трафик.

В таблице подробно сравниваются SG и NACL.

SG	**NACL**
Работает на уровне инстансов (первый уровень защиты)	Работает на уровне подсети (второй уровень защиты)
Поддерживает только разрешающие правила	Поддерживает разрешающие и запрещающие правила
С сохранением состояния: любые изменения, применяемые к входящему правилу, будут автоматически применяться к исходящему правилу	Без сохранения состояния: любые изменения, примененные к входящему правилу, не будут применяться к исходящему правилу
Мы оцениваем все правила, прежде чем принять решение, разрешать или нет трафик	Мы обрабатываем правила поочередно в порядке их следования, решая, разрешать или запрещать трафик
Применяется только к инстансу в случае, если кто-то указывает группу безопасности при запуске инстансов или позже связывает ее с инстансом	Автоматически применяется ко всем инстансам в подсетях, с которыми он связан (резервный уровень защиты, поэтому вам не нужно полагаться на то, что кто-то укажет правильную группу безопасности)

Как показано на схеме на рис. 6.2, группы безопасности следует применять к группам инстансов, использующих одинаковые порты для связи на транспортном уровне *(уровень 4)*, тогда как NACL нужно задействовать, чтобы предотвратить подключение между двумя большими масками подсети.

В этом примере мы хотим, чтобы пользователи подключались (через Интернет) к подсети *1*, но не к подсети *2*. NACL лучше всего подходят для обеспечения соблюдения этого правила на уровне подсети, тогда как конкретные порты приложений будут установлены на уровне группы безопасности.

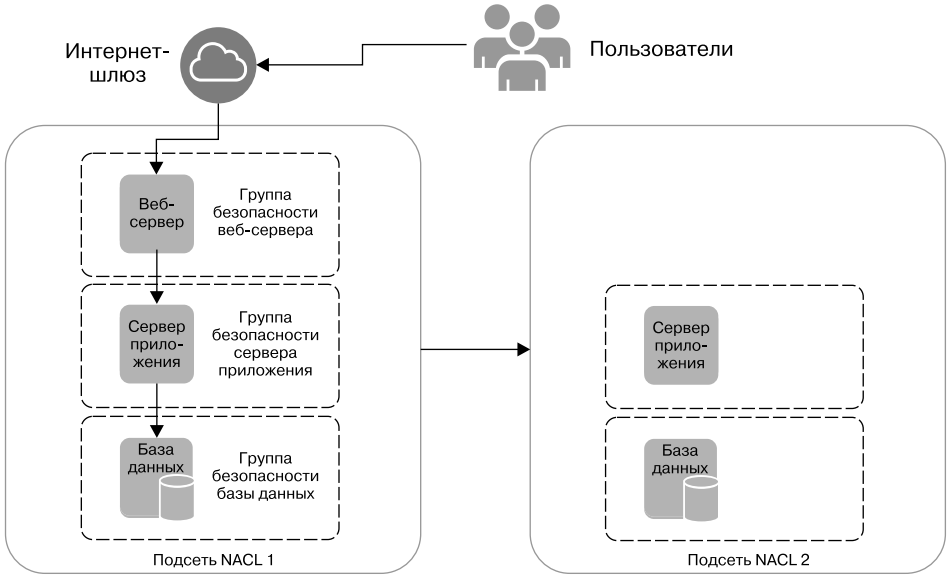


Рис. 6.2

При определении того, как использовать SG и NACL в своем развертывании, на этапе проектирования учитывайте следующие факторы.

- Как мой шаблон подсети выглядит в плоскости сети?

  Пример: нужны частные и общедоступные подсети, чтобы изолировать мои локальные приложения от доступа из Интернета.

- Сгруппированы ли основные серверы в определенные подсети?

  Пример: подсети веб-уровня, уровня приложений и уровня БД.

- Какие кластеры или подсети виртуальных машин смогут выходить за пределы частной сети, а какие — нет?

  Даже в сети без какой-либо иерархии будут встречаться похожие виды трафика, которые можно сгруппировать соотетствующим образом.

- Какие порты и протоколы используются моими приложениями?

  Понимание общих шаблонов трафика приложений поможет создать соответствующие правила брандмауэра.

- Нужно ли вашим приложениям выходить в общедоступное сетевое пространство?

  Разберитесь, какие приложения должны обращаться к общедоступным ресурсам для получения исправлений/обновлений и к чему они должны подключаться.

Ответив на эти вопросы, инженер по безопасности и сетям сможет заложить более надежную основу для обеспечения масштабируемого и безопасного состояния брандмауэра для облачной среды.

Журналы и мониторинг

На облачных платформах есть несколько типов собственных журналов, позволяющих пользователям создавать обширный поток подробных журналов для аудита и мониторинга. Несмотря на обширное и постоянно растущее количество предложений в этой сфере, все еще существуют пробелы, в первую очередь в области ведения журналов приложений. Мы рассмотрим методы устранения этих пробелов с помощью решений с открытым исходным кодом или продуктов сторонних разработчиков.

Сетевые журналы

Сетевые журналы позволяют пользователю облака просматривать сетевой трафик на уровне своей частной облачной сети. Журналы потоков AWS VPC, журнал потоков Azure и GCP Stackdriver — это нативные сервисы, которые работают на соответствующих облачных платформах и предоставляют инструменты ведения журнала и анализа сети.

Журналы потоков содержат большое количество информации, включая перечисленную далее, но не ограничиваясь ею:

- IPv4- или IPv6-адрес отправителя и получателя;
- порт, используемый приложением на стороне отправителя и получателя;
- номер протокола по версии IANA;
- количество пакетов;
- размер в байтах;
- время начала и окончания захвата трафика;
- действие/состояние (был трафик разрешен правилами брандмауэра или нет);
- ID аккаунтов;
- ID интерфейса (логический идентификатор виртуального сетевого интерфейса).

Эти журналы хранятся в облачном сервисе, но могут быть экспортированы в другие целевые хранилища, что позволяет пользователям облака ассимилировать и компилировать множество различных источников журналов в одном центральном репозитории.

Журналы аудита

Журналы аудита позволяют фиксировать и сохранять все управленческие/логические события, администрируемые в облачной среде. AWS CloudTrail, журналы управления Azure Control и GCP Cloud Audit Logging — все это нативные сервисы, которые предоставляют журналы административного аудита, чтобы понять, как пользователи взаимодействуют с облачной средой.

Для создания проверенных сред (GxP, HIPAA) или просто поддержания масштабируемой и устойчивой корпоративной облачной среды очень важно иметь возможность отслеживать действия сотен пользователей, которые могут взаимодействовать с облачной средой. Нативные журналы аудита позволяют делать это.

Поскольку все сервисы облачных провайдеров основаны на масштабируемых API, все взаимодействия с облачной средой можно свести к API-вызовам (даже изменения, сделанные с применением графического интерфейса). Журналы, создаваемые такими API-вызовами, содержат множество информации, включая следующую, но не ограничиваясь ею:

- тип события, имя, время и источник;
- ID ключа доступа;
- ID аккаунта;
- имя пользователя;
- область;
- IP-адрес источника;
- параметры запроса;
- элементы ответа.

Эти журналы хранятся в целевых облачных хранилищах или в самих облачных сервисах, но могут быть экспортированы и объединены с другими журналами для централизации, что позволяет пользователям видеть множество учетных записей.

Инструменты мониторинга

Инструменты мониторинга предоставляют пользователю облака панель мониторинга для просмотра показателей и изучения поведения облачной среды с течением времени. Думайте об этих сервисах как об облачных инструментах мониторинга облачной инфраструктуры, которые легко интегрируются с другими облачными сервисами. AWS CloudWatch, Azure Monitor и мониторинг GCP Stackdriver — все это нативные сервисы, которые работают на соответствующих облачных платформах и обеспечивают аналогичные уровни функциональности.

На ранних этапах цикла разработки этих сервисов они сразу интегрировались с другими облачными сервисами. Сейчас, по прошествии некоторого времени и после добавления дополнительных функций, эти сервисы могут обрабатывать пользовательские показатели, позволяя пользователям облака передавать настроенные ими параметры приложений в сервисы мониторинга. Команды, занимающиеся безопасностью и администрированием, всегда должны стремиться к тому, чтобы система мониторинга всего стека была доступна в виде единой консоли управления.

Некоторые важные функции, которые следует применять с этими сервисами:

- ежеминутный детальный мониторинг;
- установка и запуск сигналов оповещений, когда показатели мониторинга достигают определенных пороговых значений;
- автоматизация действий на основе сигналов оповещений;
- создание собственных информационных панелей с графиками и статистикой.

В отличие от других решений для мониторинга, сервисы облачного мониторинга всегда включены, что дает явное преимущество в производительности и доступности, поскольку они работают в распределенном режиме и ими управляют облачные провайдеры. Эти сервисы интегрируются с другими облачными сервисами, повышая функциональную совместимость. Файлы журналов могут быть зашифрованы, что снижает риск взлома с использованием одного из самых распространенных видов атак.

Управление конфигурацией

Поскольку все административные операции в облачной среде (через графический интерфейс или интерфейс командной строки) являются API-вызовами, отслеживание состояний и изменений конфигурации становится выполнимой задачей. Такие сервисы, как AWS Config, позволяют пользователям отслеживать шаблоны развертывания, сохранять состояния конфигурации и следить за тем, как со временем изменяются облачные ресурсы.

Пользу, которую это приносит для обеспечения безопасности, трудно переоценить. Возможность создать реестр облачных ресурсов, а также конфигурации программного обеспечения, выполняемого внутри виртуальных машин, дает пользователям уверенность в том, что они способны обеспечить безопасность и выявить любую рассинхронизацию конфигурационных файлов. Администраторы могут оповещаться о каждом вносимом изменении с помощью механизмов уведомлений. Дополнительным преимуществом становится возможность аудита и оценки соответствия облачной среды.

Управление идентификацией и доступом

Вероятно, самый фундаментальный и важный сервис, когда речь идет о безопасности облачной среды, — *IAM (identity and access management — управление идентификацией и доступом)*. Он позволяет пользователям облака давать пользователям, машинам и другим сервисам доступ для изменения, эксплуатации облачных сервисов и работе с ними, а также управления ими, масштабируемым и безопасным способом. Все основные поставщики облачных услуг имеют надежный сервис IAM, который интегрируется с зонами доступности и другими решениями идентификации LDAP.

Компоненты сервиса IAM можно разбить на следующие основные составляющие.

- *Пользователи*. Сущность, созданная для представления человека или сервиса, которые используют ее для взаимодействия с облачной средой. Пользователь определяется по имени и имеет набор учетных данных для доступа к облаку (секретный ключ и ключи доступа).
- *Роли*. Сущность, отражающая тип доступа, который может потребоваться отдельному лицу, машине или сервису. Роль похожа на пользователя, но предполагается, что ее может принять на себя любой, кому требуются привилегии роли.
- *Политики*. Это правило, реализованное в коде, которое явно или неявно предоставляет доступ к облачным сервисам, ресурсам или объектам либо запрещает его. Политики привязаны к ролям или пользователям, они разрешают или ограничивают их действия в среде.
- *Временные учетные данные безопасности*. Сервис, предоставляющий временные учетные данные с ограниченными привилегиями для пользователей IAM, которые аутентифицированы/объединены сервисом IAM.

Аналогичным образом облачные провайдеры разработали надежные сервисы для аутентификации и авторизации пользователей в мобильных приложениях, известные как authN и authZ. AWS Cognito, сервис мобильных приложений Azure и GCP Firebase — примеры таких сервисов. Они позволяют разработчикам интегрировать масштабируемый и легкодоступный сервис authN/authZ в свои мобильные приложения, которые могут легко интегрироваться в их облачную среду на стороне сервера.

Сервисы и модули шифрования

Гипероблака предлагают полностью управляемые сервисы, которые генерируют ключи криптографического шифрования и управляют ими. Эти ключи можно использовать для шифрования данных, принимаемых и хранимых в других облачных сервисах, или данных, хранящихся на уровне приложения. Физическая безопасность, техническое обслуживание оборудования и доступность ключей управляются как сервис, позволяя пользователям сосредоточиться на том, как

применяются ключи, и обеспечивать надлежащую защиту своих данных. AWS предлагает сервис управления ключами *Key Management Service (KMS)*, Azure Key Vault и сервис управления ключами GCP.

Ключи генерируются в соответствии со спецификацией Advanced Encryption Standards 256 бит (AES-256), установленной Национальным институтом стандартов и технологий США (US NIST). Они соответствуют передовым стандартам, таким как Федеральные стандарты обработки информации 140-2 (FIPS 140-2), что позволяет шифровать конфиденциальную информацию, требующую высокого уровня безопасности шифрования, с помощью этих облачных сервисов. Использование данных сервисов — ключ к реализации в облаке масштабируемых совместимых архитектур, о чем мы расскажем далее в этой главе.

Брандмауэры веб-приложений

Существуют облачные сервисы, предоставляющие возможности *брандмауэра веб-приложений (web application firewall, WAF)* под ключ. Они позволяют создавать правила, которые могут фильтровать веб-трафик на основе условий, включающих IP-адреса, заголовки и текст HTTP или настраиваемые URI. Эти сервисы упрощают развертывание масштабируемого и легкодоступного слоя защиты от атак, направленных на задействование уязвимостей в пользовательских или сторонних веб-приложениях, которые вы можете развернуть. А еще они упрощают создание правил, которые могут противодействовать распространенным эксплойтам, таким как SQL-инъекции и межсайтовый скриптинг. Можно также настроить правила для блокировки на основе GeoIP, чтобы ограничить доступ из конкретных стран.

Помимо настраиваемых наборов правил, которые могут конфигурировать пользователи, через нативные сервисы WAF доступны управляемые наборы правил. Эти наборы ежедневно обновляются, чтобы адаптироваться к новым CVE и угрозам, обнаруженным командами, ответственными за облачную безопасность. Это позволяет вашему отделу безопасности пользоваться наработками глобальных, узкоспециализированных команд облачного провайдера.

И AWS WAF, и Azure WAF изначально предоставляют эти возможности на своих платформах. AWS WAF также поддерживается через API, что открывает новые возможности для интеграции в полный процесс DevSecOps.

Соответствие

Соответствие требованиям — побочный продукт обеспечения безопасности в целом. Соответствующая среда, как правило, безопасна, поскольку она придерживается передовых практик, регулирующих использование, обслуживание и работу набора данных или среды. Для достижения соответствия администраторы должны предоставить документацию и контрольные журналы и продемонстрировать

средства операционного контроля сторонним аудиторским организациям. В облачной среде разделение обязанностей по обеспечению безопасности между поставщиком облачных услуг и их потребителем означает, что обе организации должны работать вместе для достижения соответствия. Облачные провайдеры делают это, предоставляя автоматизированные инструменты для доступа и создания отчетов о соответствии для части модели безопасности облачных провайдеров. Инструмент AWS Artifact, Microsoft Trust Center, а также страница Google Cloud Compliance предоставляют удобные средства для просмотра и загрузки копий различных свидетельств соответствия. К ним относятся большой и постоянно растущий список отчетов о соответствии, таких как стандарт управления безопасностью (ISO 27001 Security Management Standard), специальные облачные элементы управления (ISO 27017 Cloud Specific Controls), стандарт безопасности данных индустрии платежных карт (Payment Card Industry Data Security Standard, PCI DSS), отчет о контроле аудита (SOC 1 Audit Controls Report), отчет о контроле соответствия (SOC 2 Compliance Controls Report), отчет об общем контроле (SOC 3 General Controls Report), положения о международной торговле оружием (International Traffic in Arms Regulation, ITAR), различные национальные законы о конфиденциальности, решения федерального совета по надзору за финансовыми учреждениями (Federal Financial Institutions Examinations Council, FFIEC), альянс облачной безопасности (Cloud Security Alliance controls, CSA), информационные службы по вопросам уголовной юстиции (Criminal Justice Information Services, CJIS) и многие другие.

Автоматизированная оценка безопасности и DLP

Облачные сервисы продвигаются все дальше по стеку (от традиционного представления «инфраструктура как сервис») и предоставляют больше инструментов, помогающих создавать и оценивать безопасность кода, развернутого клиентами. Эти сервисы помогают выявлять отклонения от передовых методов обеспечения безопасности в приложениях до и во время развертывания. Их можно интегрировать в ваш процесс DevOps для автоматизации отчетов об анализе безопасности по мере продвижения по конвейеру развертывания. AWS Inspector, облачные агенты Azure Security Center Qualys и Cloud Security Scanner предоставляют некоторые или все функции, упомянутые ранее.

Двигаясь дальше по стеку, облачные сервисы предоставляют пользователям возможность обнаруживать, классифицировать и защищать конфиденциальные данные, хранящиеся в облачных средах. Эти сервисы действуют на основе машинного обучения для классификации данных в крупных масштабах без надзора со стороны человека. Постоянно отслеживая облачную среду, администраторы могут быть уверены, что автоматически обнаруживаются критически важные для бизнеса данные, такие как PII (личная информация), PHI (защищенная медицинская информация), ключи API и секретные ключи, и на основе уве-

домлений принимаются соответствующие меры. AWS Macie, Azure Information Protection и Google Cloud Data Loss Prevention API позволяют пользователям обнаруживать и редактировать конфиденциальные данные внутри своих сред.

Методы обеспечения облачной безопасности

Теперь, когда мы получили подробное представление о встроенных средствах безопасности, доступных в облаке, можем изучить несколько примеров того, как проектировать и создавать общие развертывания. Важно понимать, что в любом отдельно взятом случае применяются не все инструменты безопасности и достигается неполное соответствие другим шаблонам безопасности. Мы стремимся продемонстрировать все возможные подходы к разным проблемам и дать читателю возможность выбирать и комбинировать различные решения.

В качестве первого примера рассмотрим трехуровневое веб-приложение (рис. 6.3).

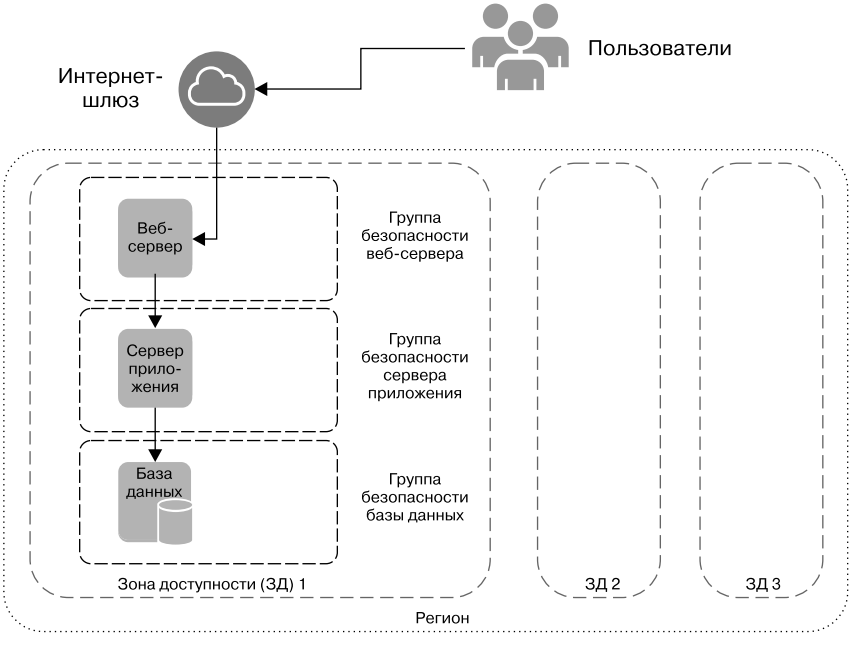


Рис. 6.3

Группы безопасности следует использовать для разделения уровней в стеке, как показано на схеме. У каждого уровня есть собственная группа безопасности, которая имеет уникальные правила, минимизирующие векторы атак.

Три уровня базового трехуровневого веб-приложения как шаблона безопасности — это сеть, приложение и база данных. Маршрутизация настроена так, что пользователь из Интернета не будет напрямую подключаться к уровням приложения или БД и взаимодействовать с ними. Уровень БД не будет взаимодействовать с веб-уровнем. Учитывая все эти особенности организации сети, мы должны соответствующим образом разработать правила нашей группы безопасности (ГБ).

Группа безопасности	Направление	Протокол	Порт	Назначение	Примечания
ГБ веб-уровня	Входящий	TCP	22, 80, 443	0.0.0.0/0	SSH, HTTP и HTTPS
ГБ уровня приложения	Входящий	TCP	22, 389	<Корпоративная сеть>	SSH, LDAP
	Исходящий	TCP	2049	10.0.0.0/8	NFS
ГБ уровня базы данных	Входящий	TCP	1433, 1521, 3306, 5432, 5439	ГБ уровня приложения	Порты по умолчанию для Microsoft SQL, Oracle DB, MySQL/ Aurora, Redshift, PostgreSQL

Мы привели примеры конфигураций для трех разных групп безопасности, соответствующих каждому уровню трехуровневого веб-приложения.

Используя три отдельные группы безопасности для каждого уровня, мы можем сделать поверхность атаки для каждой группы инстансов как можно меньшей, сохранив при этом функциональность. После настройки всех групп безопасности и обеспечения работы стека необходимо задействовать методы и подходы безопасности, описанные ранее в данной главе. Для поддержания безопасной среды нужно обеспечить, чтобы конфигурация не отклонялась от исходной структуры.

На данном этапе в игру вступает управление конфигурацией сервисов. С помощью либо настраиваемых сценариев (с использованием API, доступных от каждого CSP), либо таких сервисов, как AWS Config, можно обнаружить изменения конфигурации и принять меры, когда они выйдут за пределы установленных границ. Это очень важно, если в производственной среде работают десятки людей. Ожидайте, что произойдут изменения, противоречащие политике, и запаситесь инструментами для их обнаружения и смягчения их влияния.

Если продолжить работу над примером трехуровневого веб-приложения, мы можем наложить слой на простой цикл WAF, реализующий расширенные функции (рис. 6.4). Можно также предусмотреть специальные URL, которые будут служить приманками для обнаружения ботнетов, захвата их IP-адресов и автоматического добавления этих адресов в правила WAF.

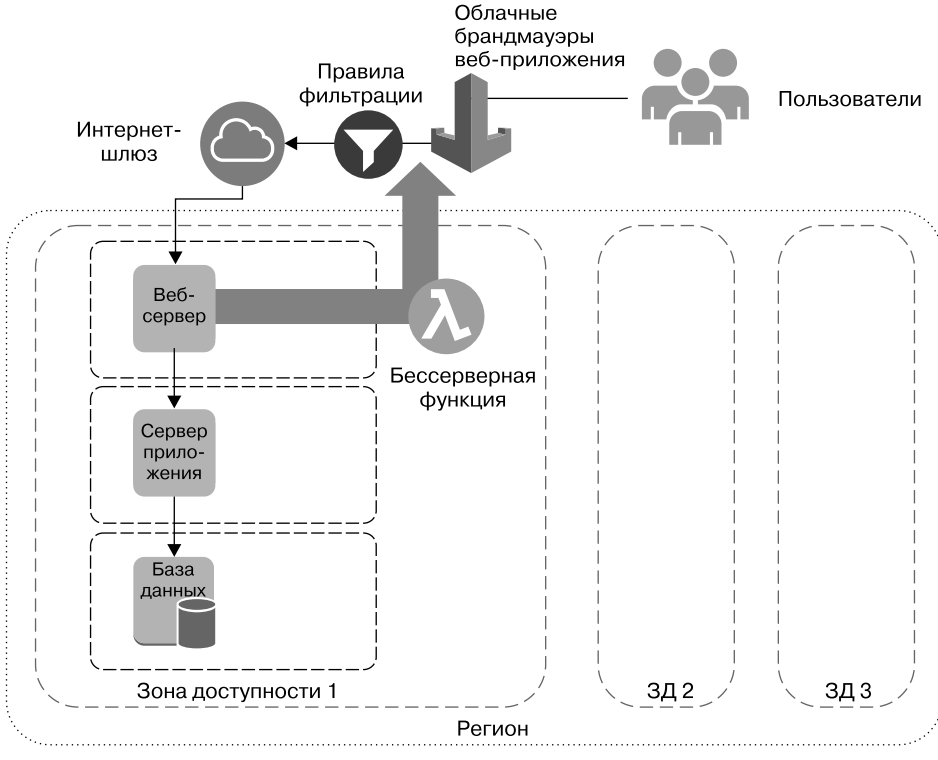


Рис. 6.4

Как показано на схеме, использование брандмауэров веб-приложений — это метод повышения безопасности в облаке. Сервисы WAF от облачных провайдеров управляются API, что позволяет создавать программные циклы, такие как сценарий автоматической ловушки.

Для этого к общедоступному сайту добавляют скрытые URL-адреса, которые невидимы для обычных пользователей, но будут обнаружены в процессе веб-скрейпинга, выполняемого ботами. После того как бот переходит по скрытым URL-адресам (honeypot), сценарий может получить его IP-адрес из веб-логов и добавить его в правила WAF для внесения этого конкретного IP-адреса в черный список. Сервис бессерверных функций под названием Lambda доступен на AWS и легко поддерживает этот подход.

Рассмотрим другой пример корпоративной среды (рис. 6.5).

Для корпоративной ИТ-среды обычно характерна сеть, распределенная по нескольким офисам и региональным облачным средам. Подход к облачной безопасности требует от нас использования тонких подходов на нескольких уровнях для уменьшения уязвимостей.

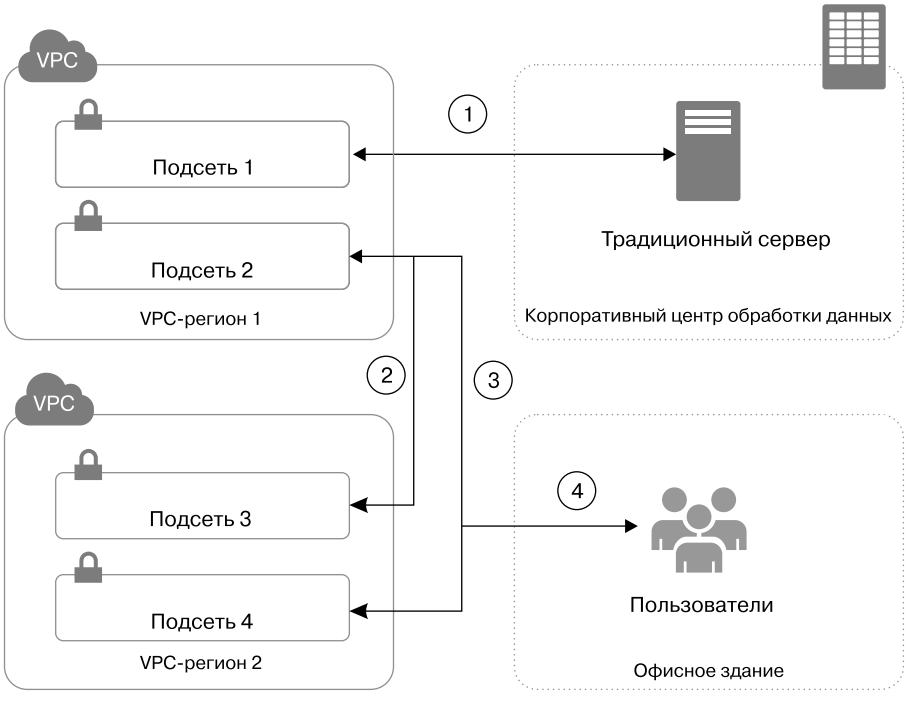


Рис. 6.5

Корпоративные среды обычно распределены по нескольким сайтам и облачным провайдерам. Это сделано для минимизации бизнес-рисков, уменьшения масштаба потенциального ущерба, поддержки систем, которые должны развертываться локально (например, приложений управления мощностью ядерных реакторов), а также для организации работы вспомогательных офисов. Это создает большую поверхность атаки и требует контроля безопасности в нескольких точках обмена сетевым трафиком.

Первые инструменты, к которым мы можем обратиться, знакомы всем специалистам ИТ-индустрии: VPN типа «сеть — сеть» и частные оптоволоконные соединения, предоставляемые интернет-провайдерами и CSP (например, AWS Direct Connect). Второй способ — использовать *списки контроля доступа к сети (Network access control lists, NACL)* для предотвращения любых нестандартных ситуаций. Например, на предыдущей схеме показаны несколько разных подсетей в двух облачных регионах плюс два корпоративных местоположения. Если у нас есть офисные приложения для повышения производительности, работающие в подсети *2*, мы можем включить NACL, чтобы разрешить только пакеты из диапазона адресов подсети офисных пользователей. Допустим, БД также находятся в подсети *2*, мы можем добавить правила, разрешающие подключение к подсетям *3* и *4* для работы с серверами приложений, работающими в регионе *2*.

Кроме того, мы можем ограничить взаимодействие кластера *высокопроизводительных вычислений (High performance compute, HPC)*, использующего *персональные данные (Personal identifiable information, PII)* и работающего в подсети *1*, для связи с локальными серверами только через сетевой путь *1*. Применение NACL позволяет обеспечить высокий уровень контроля подключений между разрозненными корпоративными сайтами.

Рассматривая каждый VPC по отдельности, мы должны понимать тип входящего и исходящего трафика. Это дает нам общее представление о том, как выглядит хороший трафик, позволяет перехватывать неподходящие пакеты, идущие по неутвержденным путям, и делает доступным инструмент для отладки неправильно работающих приложений. У CSP есть такие сервисы, как AWS VPC Flow Logs, которые собирают и создают журналы, содержащие IP-адрес источника/назначения, размер пакета, время и другие показатели для пакетов, входящих в VPC и исходящих из него. Эти журналы могут быть загружены в инструменты SIEM для отслеживания несогласованного или ненормального поведения (рис. 6.6).

Рис. 6.6

Облачно-ориентированный подход к безопасности выражается в использовании многослойных защитных сервисов на каждом уровне стека, от нижнего сетевого уровня до сетевых интерфейсов в ОС инстанса, работающих с данными, хранящимися в среде, и, наконец, с поведением пользователей в общей среде.

Помимо разветвленности сети, которую нам необходимо защищать, следует также принимать во внимание тот факт, что с нашей средой будут взаимодействовать сотни и тысячи пользователей. Как безопасно и ответственно подойти к созданию условий для их продуктивной работы, минимизировав при этом вероятность выполнения ошибочных действий?

Идентификация

Облачная система управления идентификацией преследует три основные цели.

- *Обеспечение продуктивности выполнения повседневных обязанностей сотрудниками организации.* К ним относятся администраторы БД, сотрудники службы безопасности и разработчики.
- *Создание условий для выполнения компьютерами автоматических действий внутри среды.* Эти компьютеры (а также функции и приложения) становятся все более важными по мере разрастания среды.
- *Разрешение пользователям/потребителям безопасно подключаться к общедоступным сервисам.* Это влечет за собой требования к защите и аутентификации личности потребителей.

У облачных провайдеров есть хорошо развитые IAM-сервисы (например, AWS IAM), которые позволяют создавать настраиваемые *политики*, определяющие, что можно делать в облачной среде. Эти политики можно привязывать к группам; в этом случае они объединяются с наиболее запретительной интерпретацией в единый орган управления. Для каждого человека, работающего в среде, можно создать *пользователя* с уникальным логином для доступа к консоли и уникальным ключом/секретом для выполнения API-вызовов в среду. Каждое взаимодействие пользователя со средой будет ограничено тем, что разрешено политиками, привязанными к группе, в которую он включен (рис. 6.7).

Политики можно применять как для ролей, так и для групп. Их можно даже прикрепить к конкретным пользователям, хотя это не лучшая практика. Сервисы и компьютеры, которым назначены роли, могут неявным образом получать доступ к другим частям облачной среды, что делает возможным использование разных методов автоматизации.

Работая с несколькими облачными учетными записями и другими платформами SaaS, рекомендуется предоставить своим сотрудникам решение *единого входа (Single Sign On, SSO)*. Это упрощает и делает более безопасным переключение профилей пользователей в нескольких облачных средах (Azure, AWS, GCP, Salesforce и т. д.).

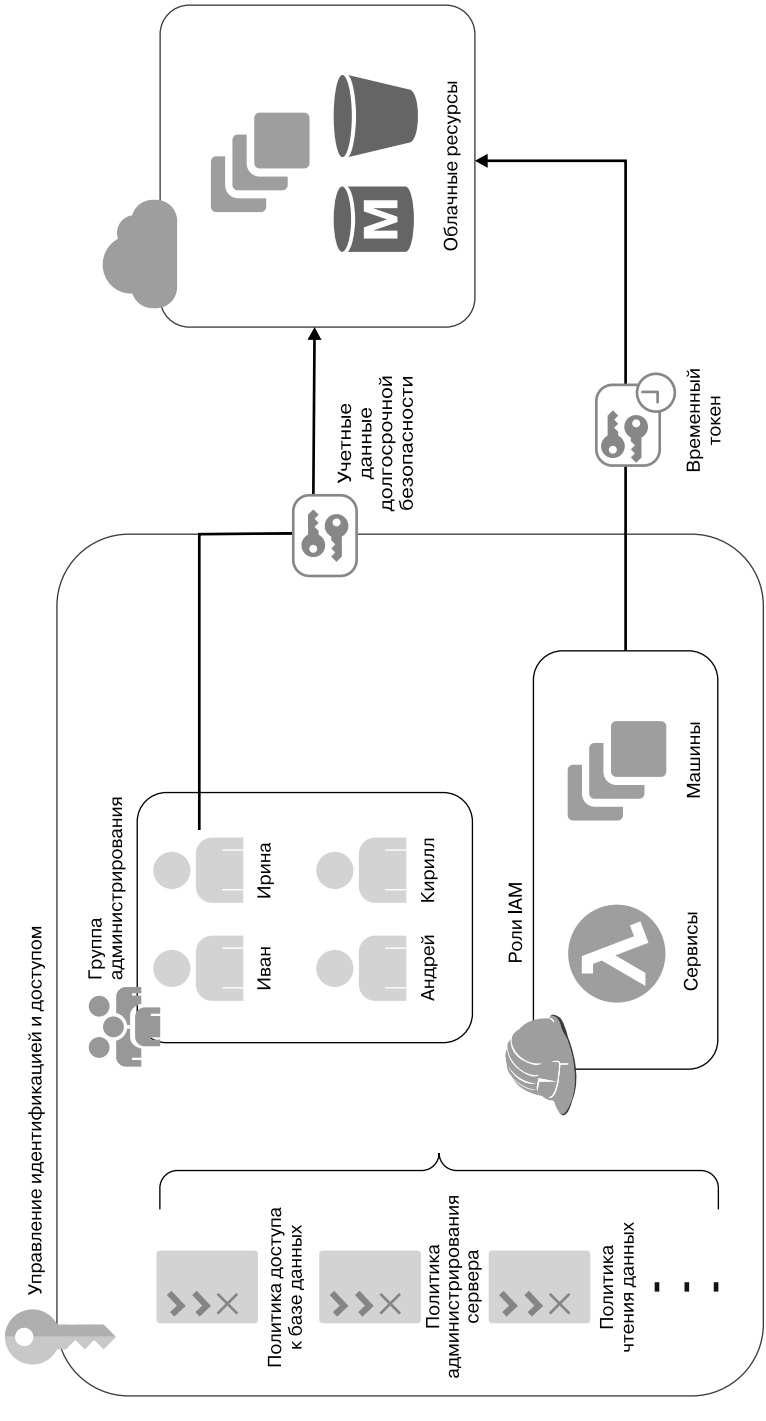

Облачные ресурсы
Учетные данные долгосрочной безопасности
Временный токен
Управление идентификацией и доступом
Группа администрирования
Иван
Ирина
Андрей
Кирилл
Роли IAM
Сервисы
Машины
Политика доступа к базе данных
Политика администрирования сервера
Политика чтения данных

Рис. 6.7

Одна из основных человеческих ошибок, допускаемых при распределении ИТ-инфраструктуры между несколькими провайдерами, заключается в том, что пользователь управляет несколькими входами в систему (обычно для одной и той же среды CSP), что может иметь катастрофические последствия (не менее неприятной является ситуация, когда при входе в систему пользователь получает доступ к чужому профилю).

Акцент в облачной среде всегда делается на автоматизации. Каждый ИТ-отдел должен быть ориентирован на то, чтобы машины могли выполнять свою работу, высвобождая пользователям время для создания большего количества машин. Чтобы обеспечить безопасную работу этого полезного цикла, нам нужны механизмы, позволяющие задействовать машины, но ограничить их использование злоумышленниками. Следует отметить, что приложения, работающие с облачными сервисами, будут использовать аналогичные механизмы, поэтому мы обсуждаем не только инструменты управления ИТ-средой, но и все приложения, программно взаимодействующие с облаком.

Первый вариант — это использование *ролей*. Роли предоставляются сервисам и машинам с определенными политиками (это те же самые политики, которые мы создали для групп/пользователей). Эти роли предоставляют возможности присоединенным объектам, которые обрабатываются облачными провайдерами. Ключи программно передаются машине и сервисам при выполнении API-вызовов.

Для приложений встраивание секретов в код — опасная уязвимость. Вместо этого мы можем использовать второй вариант — токен-сервисы, которые генерируют и передают ограниченные по времени учетные данные для выполнения API-вызовов. Срок действия такого ключа истекает в конце установленного времени, после чего приложению необходимо будет сделать новый запрос к сервису токенов. AWS Security Token Service (STS) — очень полезный инструмент, позволяющий использовать секреты в приложениях.

Мобильная безопасность

Интернет становится все более мобильным — большая часть веб-трафика теперь идет с мобильных устройств, и мобильные приложения доминируют в сфере разработки. Подход к безопасности в облаке должен учитывать безопасность данных, созданных в этих приложениях, и безопасность потока данных между облачными системами и конечными пользователями.

В прошлом разработчики мобильных приложений управляли ключами доступа с помощью пользовательских реализаций кода или сторонних инструментов, приобретенных на индивидуальной основе или по лицензии. Это не только увеличивало накладные расходы для команд разработчиков, но и означало, что они должны поддерживать и обновлять этот код для устранения новых уязвимостей. Вместо того чтобы управлять серверной инфраструктурой этих функций в самих приложениях, мы можем воспользоваться сервисами CSP, которые позволяют делать это эффективно и с поддержкой масштабирования.

Такие сервисы, как AWS Cognito, помогают разработчикам и архитекторам решать некоторые из описанных проблем. Первым делом необходимо определить пользователей и группы, которые будут взаимодействовать с вашим приложением. Вы можете настроить и определить атрибуты, необходимые для идентификации уникальных пользователей. Сложность пароля, длину, применение специальных символов и условий регистра можно задать, чтобы обеспечить базовый уровень безопасности. Сервис поддерживает многофакторную аутентификацию (MFA) с помощью электронной почты и SMS, заботится он и об инфраструктуре обмена SMS.

Кроме того, эти облачные сервисы упрощают интеграцию приложений-клиентов. Поддерживая стандарты OAuth 2.0, такие сервисы, как AWS Cognito, могут выдавать токены доступа конечным пользователям, предоставляя доступ к защищенным ресурсам, поддерживающим интерфейс вашего приложения (рис. 6.8).

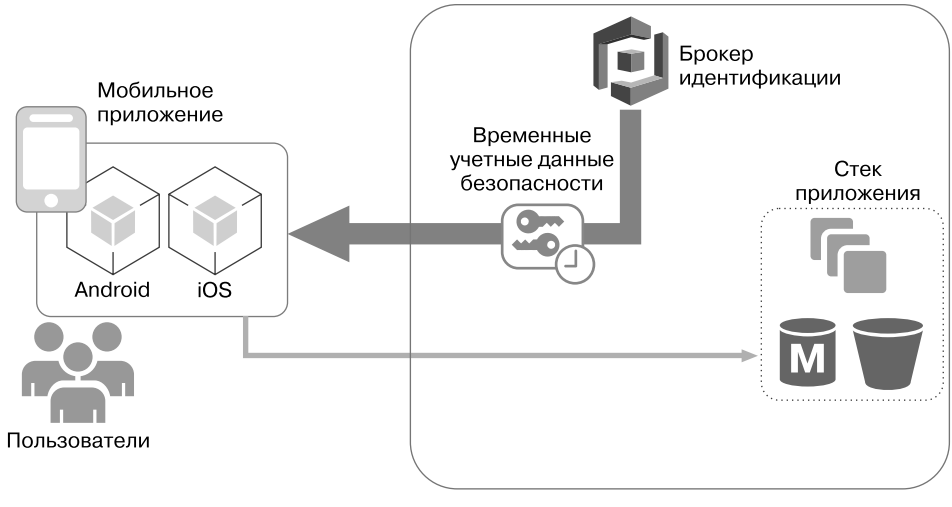


Рис. 6.8

Облачная система управления идентификацией мобильных приложений, созданная с помощью облачных сервисов, помогает облегчить тяжелую работу по созданию масштабируемых и безопасных сервисов. Эти сервисы используют общие стандарты, такие как OAuth 2.0, для делегирования доступа к ресурсам, которые нужны вашему приложению.

DevSecOps

Термин DevSecOps, который все чаще используется в отрасли, означает конвергенцию разработки, безопасности и эксплуатации. Поскольку методы DevOps широко распространены и приняты во всех технологических практиках, гибкие методики, практикуемые в DevOps, делают недостаточный акцент на безопасности.

DevSecOps применяет тот же гибкий принцип «сам создаю, сам обслуживаю» к безопасности, перенося ее в контекст непрерывной интеграции/развертывания. В конечном итоге это вера в то, что за безопасность отвечает определенный набор компонентов или небольшая команда. Это сочетание инструментов, платформы и мировоззрения, а также мысли о том, что каждый несет ответственность за безопасность и должен внедрять передовые методы безопасности на любом этапе жизненного цикла разработки/развертывания/эксплуатации.

У DevSecOps есть руководящие принципы, составляющие облачно-ориентированный подход, и их прекрасно демонстрирует манифест DevSecOps (рис. 6.9).

- **Прислушивайтесь,** а не всегда говорите «нет».
- **Наука о данных и безопасности** выше страха, неопределенности и сомнений.
- **Открытый вклад и совместная работа**, а не требования исключительно к безопасности.
- **Потребительские услуги безопасности с API** вышеутвержденных средств контроля безопасности и бумажной работы.
- **Бизнес-ориентированные оценки безопасности** вместо шаблонной безопасности.
- **Тестирование на уязвимости командой Red&Blue**, а не простое сканирование и теоретические уязвимости.
- **Круглосуточный упреждающий мониторинг безопасности** важнее реагирования после получения информации об инциденте.
- **Общая информация об угрозах** важнее сохранения информации при себе.
- **Операции соответствия** важнее папок и чек-листов

Рис. 6.9

Подобно тому, о чем говорилось в главе 5, безопасность выигрывает от восприятия всего как кода, что называется здесь *«безопасность как код» (SaC)*. Это означает все тот же подход IaC и его применение к обеспечению безопасности и эксплуатации. Для конфигурации систем контроля доступа, брандмауэров и групп безопасности можно использовать шаблоны или рецепты. Эти шаблоны должны использоваться в организации как отправная точка для создания среды. Их также можно задействовать в качестве ориентиров для отслеживания отклонений от утвержденных схем.

Инструменты для обеспечения безопасности в облаке

Теперь, когда у нас есть четкое представление о подходах и стратегиях создания безопасных и надежных облачных решений, представим вспомогательные инструменты, продукты и проекты с открытым исходным кодом. Следует отметить, что эти инструменты помогают достичь целей, обсуждаемых в этой главе, но имеют

свою цену. Естественно, лучше всего самостоятельно разработать эти функции, адаптированные к вашему облаку, но это может оказаться непростой задачей.

Okta

В Okta увеличивается количество сервисов, связанных с идентификацией, но данный инструмент наиболее известен своим сервисом единого входа, который позволяет пользователям управлять логинами и паролями для большого количества отдельных учетных записей (через IaaS, PaaS и Saas). Это обеспечивает более эффективный вход в систему и помогает пользователям (особенно администраторам) управлять несколькими пользователями/ролями в рамках одной облачной учетной записи (www.okta.com/).

Centrify

Centrify — еще один популярный инструмент управления идентификацией и доступом, который интегрируется с облачными средами. Он поддерживает интеграцию с зонами доступности и позволяет автоматически создавать учетные записи. Это очень помогает в безопасной работе в масштабе крупных предприятий, управляющих десятками или сотнями облачных учетных записей (www.centrify.com/).

Dome9

При устранении неполадок в ходе масштабного развертывания того, что мы обсуждали в данной главе, может оказаться сложно визуализировать и понять истинные состояния конфигурации. Dome9 помогает решить эту проблему, создавая подробные карты групп безопасности, NACL и машин. Кроме того, Dome9 может интегрироваться в рабочие процессы DevOps, чтобы сканировать и помечать шаблоны IaC (например, *cloudformation*) на предмет обнаружения антишаблонов (dome9.com/).

Evident

Evident предлагает ряд продуктов для различных облачных провайдеров. Платформа Evident Security Platform (ESP) позволяет пользователям получить консолидированный обзор всего облачного ландшафта (для нескольких учетных записей) и помогает им находить угрозы путем управляемого реагирования на инциденты. Кроме того, ESP интегрируется с собственными облачными сервисами, такими как AWS Config, чтобы помочь обнаружить рассинхронизацию состояния с течением времени (evident.io/).

Резюме

В этой главе мы обсудили, как область безопасности должна адаптироваться к облачному миру. Прежние процессы обеспечения безопасности во многом устарели, необходим новый подход. Мы обсудили представление о применении безопасности на каждом уровне и о том, какие важные функции составляют безопасный облачный стек. Они совместимы, зашифрованы с помощью масштабируемых/доступных ресурсов шифрования, используют технологию DLP для автоматического предотвращения раскрытия данных, применяют кэширование, чтобы нельзя было записать пользовательские данные в память приложения, и, наконец, имеют защищенные учетные данные и конечные точки, позволяющие людям безопасно взаимодействовать со стеком.

Мы представили ряд сервисов облачной безопасности, которыми сегодня пользуются облачные провайдеры. К ним относятся группы безопасности, NACL, сервисы предотвращения потери данных, управления идентификацией и доступом, создания журналов и сбора данных, мониторинга, управления конфигурацией, сервисы и модули шифрования, брандмауэры веб-приложений и автоматизированные инструменты оценки безопасности.

Все эти сервисы следует объединить, чтобы на их основе разработать безопасный облачный стек. Мы рассмотрели несколько новых примеров того, как эти сервисы могут применяться в различных ситуациях, от управления IAM и конфигурации брандмауэра до управления мобильной безопасностью.

Наконец, были представлены сторонние инструменты, доступные на рынке, с помощью которых можно сократить разрыв между нынешними подходами к безопасности и подходами в облаке, обсуждаемыми в данной главе.

7 Оптимизация затрат

В этой главе мы поговорим о том, как преподносить и доносить тот факт, что использование облачных платформ выгодно с финансовой точки зрения. Прочитав эту главу, вы получите четкое представление о существующих на данный момент моделях ценообразования в облаке и о том, как их оптимизировать. Компании по-разному смотрят на эти модели, поэтому мы обсудим, какие аспекты моделей нравятся определенным лидерам бизнеса.

Когда у вас будет четкое понимание модели ценообразования, мы перейдем к формулированию экономических бизнес-обоснований для внедрения в облако. На нескольких примерах мы покажем экономическую выгоду и гибкость, которые может предложить облако. Наконец, рассмотрим доступные общие наборы инструментов и сервисы, позволяющие обеспечить мониторинг, контроль и оптимизацию затрат в облачной среде.

Прежде чем начать, определим *затраты* как *расходы, которые несет бизнес при выводе продукта или сервиса на рынок*. *Цена* будет определяться как *сумма, которую организация (потребитель или бизнес) платит за продукт или сервис*. Наконец, *альтернативная стоимость* — это *выгода, которую организация (потребитель или бизнес) могла бы получить, но отказалась от нее в пользу другого плана действий*. Иными словами, это альтернатива, которую можно выбрать, принимая решение.

Прежде чем перейти к облаку

Прежде чем мы сможем обсудить стоимость создания облачных приложений, нужно вернуться назад и изучить инфраструктуру, существовавшую до появления облака, а именно локальные системы. Когда предприятие считает необходимым создать или расширить свою ИТ-инфраструктуру, оно должно предпринять ряд действий в отношении следующих аспектов.

- *Физическое пространство*. Поиск места в существующем центре обработки данных или покупка/аренда нового пространства для создания еще одного центра.

- *Электроэнергия*. Ее легко упустить из виду. Для развертывания крупномасштабного массива серверов требуются проложенные кабели электропитания, а их на новом месте может не быть или они могут оказаться недоступными. Кроме того, запас мощности и резервные генераторы энергии часто требуются для того, чтобы в случае отключения электроэнергии или аварии не прерывались критически важные операции.
- *Физическая безопасность*. Имеет первостепенное значение при развертывании на предприятии. Для обеспечения физической безопасности оборудования, поддерживающего центр обработки данных, требуются точки доступа с ключом/пропуском, персонал службы безопасности, камеры и другая техника.
- *Возможность подключения к сети*. В зависимости от выбранного участка широкополосного подключения может не быть или оно может иметь недостаточную пропускную способность для поддержания стабильной работы центра обработки данных. В большинстве случаев центрам обработки данных требуется физически избыточное сетевое соединение для поддержки сценариев переключения при отказе сети. Для этого могут понадобиться земляные работы — рытье траншей для прокладки дополнительной линии со стороны *провайдера интернет-услуг (Internet service provider, ISP)*, что может занять месяцы (в худшем случае требуются разрешения от местных и региональных властей, чтобы проложить одну ветку сети).
- *Охлаждение*. Вычислительное оборудование сильно нагревается, и для поддержания оптимальной производительности ему необходимо работать в определенном диапазоне температур. В новейших центрах обработки данных, построенных в странах с холодным климатом, есть пассивные системы охлаждения, но подавляющему большинству требуются большие встроенные воздуховоды для отвода отработанного тепла от компьютеров и подачи холодного воздуха.
- *Физическое оборудование*. После решения вопросов, обозначенных в предыдущих пунктах, необходимо заказать, доставить, разместить в стойке, подключить, протестировать и ввести в эксплуатацию фактическое вычислительное хранилище и сетевое оборудование, которое определяет бизнес-ценность центра обработки данных. Это потребует огромного труда сотрудников центра обработки данных — десятков человеко-часов на стойку.
- *Персонал*. Пока все перечисленные задачи решаются, специализированная компания должна нанять и обучить большое количество сотрудников для проектирования, заказа, установки, тестирования и эксплуатации всех упомянутых элементов.

Неудивительно, что строительство центров обработки данных планируется на месяцы и годы вперед. Требуются крупные капитальные вложения, чтобы собрать необходимые команды, которые могут завершить сборку и подготовиться к производству.

Как узнать стоимость облака

Появление облаков почти устранило ограничения, упомянутые в каждом из пунктов перечня, приведенного в предыдущем разделе. Облачные провайдеры теперь берут на себя *недифференцированную рутинную работу* своих клиентов. Они купили физическое пространство для размещения центров обработки данных, развернули сетевые, электрические и охлаждающие системы для поддержки миллионов машин, следят за безопасностью физической инфраструктуры, выполняют фоновые проверки и устанавливают разделение обязанностей, чтобы гарантировать, что ни один сотрудник не имеет физического и логического доступа к системам. Также у них имеется штат инженеров по эксплуатации для обслуживания огромного парка машин центров обработки данных. Более подробное обсуждение масштабов облачных провайдеров см. в главе 5.

В каждом из этих пунктов операторы связи суммируют и сокращают затраты за счет экономии на масштабе. Такие компании, как AWS или Azure, могут тратить больше времени на разработку более безопасных систем, чем другие предприятия, потому что у них больше ресурсов и денег для обеспечения безопасности и они принимают на себя более серьезные обязательства. Операторы облачного провайдера могут внедрять передовые инженерные ноу-хау для разработки более экологичных и эффективных систем охлаждения, чем любое другое предприятие на рынке. Например, Google использует морскую воду для охлаждения своих центров обработки данных в Финляндии, что снижает нагрузку на местные источники пресной воды и электростанции, а также повышает эффективность. Вы можете обратиться к www.wired.com/2012/01/google-finland/, чтобы больше узнать об этом.

Помимо ощутимых преимуществ внедрения облака, существуют и нематериальные выгоды (альтернативные издержки), которые перечислены далее.

- *Снижение сложности*. Переход в облако освобождает от недифференцированной рутинной работы по управлению ИТ-инфраструктурой вашего бизнеса. Длительные переговоры о ценах на лицензии не нужны, поскольку облачный провайдер предварительно обсудил соглашения о массовом лицензировании с поставщиками. В других случаях облачный провайдер предоставляет на выбор стороннее программное обеспечение, доступное на платформе, по расценкам, включенным в стоимость вычислительных ресурсов. Наконец, облако дает вашему бизнесу возможность направить человеческий капитал, ранее связанный с управлением этой сложной сферой, на решение более важных бизнес-задач. Такое снижение сложности прямо или косвенно экономит финансовые средства.
- *Эластичная емкость*. Переход в облако дает предприятию беспрецедентную возможность расширять или сокращать свою ИТ-инфраструктуру на лету. Эта эластичность позволяет предприятиям добиваться успеха и быстро сокращать свои ресурсы после неудачного эксперимента, а также учитывать сезонные потребности бизнеса или ежедневные колебания количества пользователей.

Предприятиям больше не нужно прогнозировать спрос на годы вперед и соответствующим образом планировать работу — они могут управлять своим бизнесом и расширять его уже сегодня, при этом работа в облаке отражает их бизнес-тенденции.

- *Повышение скорости выхода на рынок*. Предприятиям, конкурирующим за лидерство на рынке, для достижения успеха требуется *гибкость*. Возможность быстрее проектировать, разрабатывать, тестировать и запускать продукты, поддерживаемые ИТ-проектами, — это доминирующий фактор, определяющий успешность компании. Внедрение облачных архитектур — ключевой фактор в увеличении скорости, с которой компания может повторять разработку продукта.
- *Глобальный охват*. Гипермасштабные облачные провайдеры имеют глобальную сеть центров обработки данных, доступных для всех клиентов. Клиенты любого масштаба могут использовать ее для глобального расширения своих продуктов или услуг за считаные минуты. Это позволяет компаниям легко выходить на новые рынки и значительно сокращает необходимое для этого время.
- *Повышенная операционная эффективность*. Каждый из основных облачных провайдеров предоставляет инструменты для сокращения времени, затрачиваемого на основные эксплуатационные процедуры и проверки, что помогает автоматизировать многие из этих функций. Ресурсы (люди, капитал и время) высвобождаются, чтобы их можно было потратить на создание продукта, который будет выделять компанию на фоне ее конкурентов. И сотрудники организации теперь могут тратить время на решение важных и сложных проблем, что дает им большее удовлетворение от работы.
- *Повышенная безопасность*. Облачные провайдеры управляют своей частью общей модели безопасности (описана в главе 6), что высвобождает ресурсы клиентов, позволяя сосредоточиться на своей части этой модели. Предоставляя собственные инструменты безопасности на платформе, облачные провайдеры расширяют и автоматизируют средства безопасности в той части модели общей ответственности, которая отводится клиенту. Они сильно влияют на моделирование и работу организации, высвобождая ресурсы, чтобы добиться большей безопасности приложений.

Эти материальные и нематериальные преимущества чрезвычайно выгодны для каждой организации, имеющей ИТ-отдел. Выгода хорошо объясняет то, что начиная с 2013 года все чаще облачные технологии внедряются на крупных предприятиях. Многие организации осознали, что у облака есть много преимуществ и почти нет недостатков — единственным допустимым исключением является нереализованная прибыль от недавних капиталовложений в ИТ-инфраструктуру. Например, компания X только что инвестировала 50 млн долларов в новый центр обработки данных, который был запущен в прошлом месяце. Если она не сможет найти способ отказаться от этого центра и воз-

местить невозвратные затраты, математические расчеты, скорее всего, будут не в пользу внедрения облака до следующего цикла обновления (этот термин применяется для описания времени между заменами компьютеров и связанного компьютерного оборудования).

Экономика облака

Во многих компаниях центральное ИТ-подразделение управляет инфраструктурой и взимает плату за ИТ-услуги (lines of business, LOB) с учетом собственных административных расходов. Это называется моделью возвратного платежа, и у нее есть свои собственные административные расходы, известные как модель возвратного платежа. Важно понимать, что величина возвратного платежа редко равна цене облака (с сопоставимым стеком технологий). В ходе анализа цен центральные ИТ-подразделения редко учитывают расходы на аренду и обслуживание помещений, обеспечение безопасности, охлаждение, счета за воду и электричество.

Цена облачного стека должна сравниваться с суммой обратного платежа, капитальных затрат (здание и оборудование), эксплуатационных расходов (электричество, охлаждение, вода), стоимости укомплектования персоналом, лицензирования (затраты на программное обеспечение виртуализации, инструменты независимых поставщиков программного обеспечения/сторонних разработчиков и т. д.), помещения, накладных и иных затрат. Это называется *совокупной стоимостью владения (Total Cost of Ownership, TCO)*. В Azure (www.tco.microsoft.com) и AWS (awstcocalculator.com) есть калькуляторы совокупной стоимости владения, которые позволяют пользователям рассчитать совокупную стоимость владения с учетом нескольких факторов, таких как зарплаты сотрудников, стоимость электричества, цена хранилищ данных, аренда и содержание помещений и т. д.).

Инструменты анализа совокупной стоимости владения помогают сравнить текущее локальное развертывание и эквивалентную среду на платформе облачного провайдера в плане реальных денежных расходов. Они не учитывают нематериальные активы (читай: альтернативные издержки), которые можно получить за счет внедрения облачной платформы.

Чтобы узнать чистую цену облачных сервисов, нужно обсудить другой набор инструментов. Существуют калькуляторы цен, доступные для основных CSP, которые позволяют с большой точностью оценить данную архитектуру на платформе CSP (если считать, что точны ваши предположения относительно использования данных, выполнения кода, размера хранилища в гигабайтах и т. д.). Простой ежемесячный калькулятор AWS (calculator.s3.amazonaws.com/index.html), калькулятор цен Microsoft Azure (azure.microsoft.com/en-us/pricing/calculator/) и калькулятор цен Google Cloud (cloud.google.com/products/Calculator/) есть в Интернете, и ими можно свободно воспользоваться.

CapEx против OpEx

При разработке бизнес-обоснований перехода в облако одним из наиболее веских аргументов для высшего руководства является возможность сделать расходы на ИТ не *капитальными затратами (CapEx)*, а *операционными расходами (OpEx)*. Капитальные затраты определяются как деньги, потраченные бизнесом на приобретение или обслуживание основных средств, таких как земля, здания и оборудование. Операционные расходы определяются как текущие расходы на ведение бизнеса/системы или предоставление услуги/продукта.

Далее приведем несколько преимуществ перехода на модель операционных расходов, которые станут для руководителей убедительным экономическим обоснованием.

- *Более низкие текущие затраты по сравнению с крупными первоначальными инвестициями.* Как упоминалось ранее в этой главе, создание центра обработки данных занимает много времени и требует значительных ресурсов для перевода вычислительной мощности в оперативный режим. В модели OpEx бизнес может достичь того же конечного результата без крупных первоначальных вложений.
- *Налоговые льготы.* Операционные расходы в сфере налогообложения рассматриваются не так, как капитальные. Бизнесу разрешается списывать операционные расходы в год, когда они были понесены, или, говоря другими словами, они могут быть полностью вычтены из налогооблагаемой прибыли бизнеса. Однако в целом капитальные расходы должны вычитаться по графику, установленному государственной налоговой инспекцией (например, IRS в США). Обычно предусматривается, что стоимость капитальных затрат может быть вычтена из налоговых расходов предприятия в течение 3–5 лет.
- *Большая прозрачность.* Облако обеспечивает высокую степень прозрачности затрат, что позволяет руководителям бизнеса обосновывать инвестиционные решения и делать из них выводы. В облачной среде можно провести убедительные рыночные тесты увеличения или уменьшения ИТ-расходов (например, мы заплатили *X* долларов за *Y* объема хранилища и времени вычислений, что привело к увеличению онлайн-дохода на *Z* процентов).
- *Амортизация капитала.* Приняв облачную модель операционных расходов, предприятия могут не только получить налоговые льготы, но и избежать амортизации своих авансовых расходов. Амортизация капитала — это постепенное снижение стоимости актива, принадлежащего компании. В ИТ-индустрии это неизбежно, поскольку на рынке всегда появляются более производительные серверы, устройства хранения и сетевые компоненты, что снижает стоимость устаревающего оборудования.
- *Более легкий рост.* Привязанные к естественной эластичности облачных ресурсов модели операционных затрат позволяют расходовать средства, соответству-

ющие естественному росту и сокращению бизнеса, в то время как инвестиции в основной капитал могут бездействовать или ограничивают возможности бизнеса определенным уровнем.

- *Отсутствие обязательств или привязки.* Беспокойство у руководителей вызывает покупка любой технологической услуги или продукта, так как это становится привязкой — договоренностью, согласно которой компания обязана иметь дело только с конкретным продуктом или услугой. Эта привязка может иметь форму эксклюзивного контракта на обслуживание или ограничения переносимости технологий (когда изменение систем для работы с другим продуктом оказывается слишком дорогостоящим). Облако в значительной степени устраняет эту ситуацию, поскольку для использования облачных сервисов не требуются предоплата или срочный контракт. (При этом с облачным провайдером можно заключить срочный дисконтный контракт. Если вы готовы взять на себя минимальную сумму затрат на платформы, облачные провайдеры будут предоставлять дополнительные скидки в течение срока действия контракта (обычно 3–5 лет). Однако вы не обязаны это делать.) Компании могут увеличить или уменьшить потребление ресурсов на платформе (даже полностью отказаться от него) без каких-либо ограничений.

Мониторинг затрат

Платформы CSP были созданы для обеспечения прозрачности затрат. Нет никаких скрытых комиссий или сборов за обслуживание, которые клиент не может увидеть, пока не начнет разработку на платформе. Все цены публикуются и постоянно обновляются на сайтах большой тройки и доступны всем, кто интересуется ценообразованием на архитектуру. Как упоминалось ранее, облачные провайдеры связи предоставляют калькуляторы цен, чтобы помочь потенциальным клиентам оценить среду перед ее созданием. Так начал делать AWS, когда выпустил свои первые облачные сервисы, и это продолжается по сей день.

Каждый облачный провайдер имеет собственные сервисы, которые помогают отслеживать, детализировать и исследовать потребление услуг и связанные с ними затраты, когда системы разрабатываются в облаке. Панель мониторинга *Billing & Cost Management Dashboard* от AWS — прекрасный пример этих облачных возможностей (рис. 7.1). Такие функции, как AWS Cost Explorer, позволяют пользователям детализировать ежемесячные счета, просматривать историю расходов и прогнозировать будущие расходы. Клиенты могут устанавливать собственные бюджеты, которые способны отправлять сигналы оповещения, когда стоимость или использование чего-либо превышает установленные лимиты. Эти инструменты позволяют администраторам облачных вычислений уверенно ограничивать использование облачных сервисов и затраты на них.

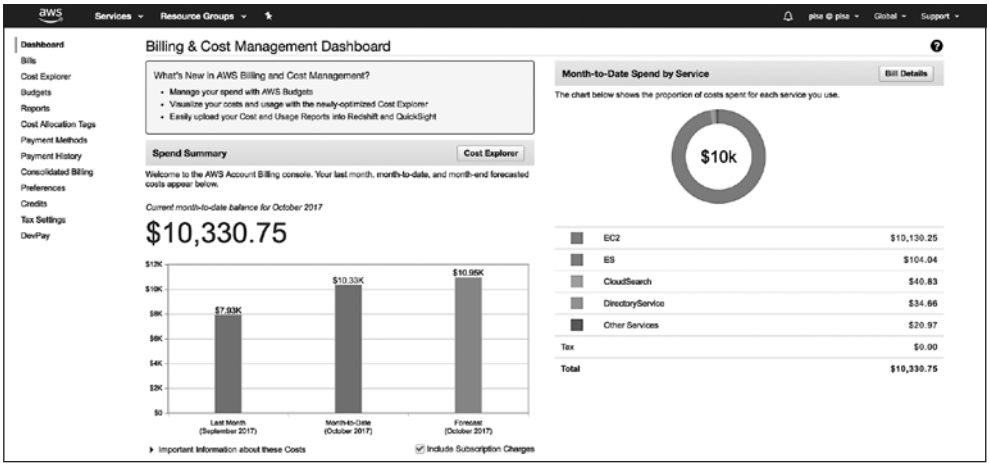


Рис. 7.1

Мы можем очень легко определить расходы за последний, текущий и следующий месяцы в одном простом представлении. Справа видно, на какие сервисы мы тратим деньги.

Практические рекомендации для архитектуры cloud native

Следующая задача администраторов облачных вычислений после установки MFA и защиты учетных корневых данных должна заключаться в ознакомлении с панелью мониторинга затрат и сервисами для выбранной облачной платформы.

Важнейшее значение для возможности навигации по тысячам или миллионам ресурсов, которые предприятие будет предоставлять в облачной среде, имеет тегирование. Тегирование — основной метод, с помощью которого ресурсы можно выделять центрам затрат, программам, пользователям, бизнес-направлениям и на какие-то определенные цели. Без тегирования практически невозможно должным образом поддерживать облачную среду в большой организации. В следующем разделе мы рассмотрим передовой опыт тегирования, связанный с управлением затратами.

Исторически на ИТ-бюджеты влияли модели капиталовложений с предварительным выделением средств. С появлением облака на первый план вышли модели операционных расходов, в которых потребители имели больший контроль над затратами. Следующий шаг эволюции — возможность устанавливать жесткие или мягкие ограничения на расходы в рамках отдельных частей или всего ИТ-ландшафта предприятия. Эти бюджеты встроены в платформу через API или сервисы уведомлений, которые предупреждают администраторов, когда потребление превышает пороговые значения.

AWS Budgets (https://aws.amazon.com/ru/aws-cost-management/aws-budgets/), Azure Spending Budgets (https://docs.microsoft.com/en-us/partner-center/set-an-azure-spending-budget-for-your-

customers) и GCP Budgets (https://cloud.google.com/billing/docs/how-to/budgets) служат примерами облачных методов для контроля и ограничения расходов. Это критически важная функция для крупных организаций, где деятельность и расходы определенных команд могут быть не сразу видны руководству или владельцам бюджета.

Практические рекомендации для архитектуры cloud native

Установите бюджеты для всех облачных учетных записей, привязанных к разным командам в вашей организации. В будущем их можно пересмотреть и изменить. Еще более важно то, что бюджеты устанавливают мягкий лимит, чтобы задать определенное поведение. Предоставлять командам карт-бланш, позволяя им оплачивать безграничные счета, — не лучшая практика. Ограничение бюджета вынудит сотрудников применять более эффективные методы работы и заставит их мыслить нестандартно и изобретательно, когда дело дойдет до проектирования систем.

Можно установить пороговые значения бюджета, чтобы предупреждать администраторов, когда расходы, использование определенных сервисов/функций или потребление предварительно оплаченных ресурсов (например, зарезервированных серверов на AWS EC2) превысят установленные лимиты. Эти бюджеты могут быть согласованы для разных периодов (ежемесячные, ежеквартальные или ежегодные), что позволит администраторам гибко устанавливать ограничения для всей организации. При желании можно настроить уведомления, предупреждающие о приближении к определенному проценту исчерпания бюджета (рис. 7.2).

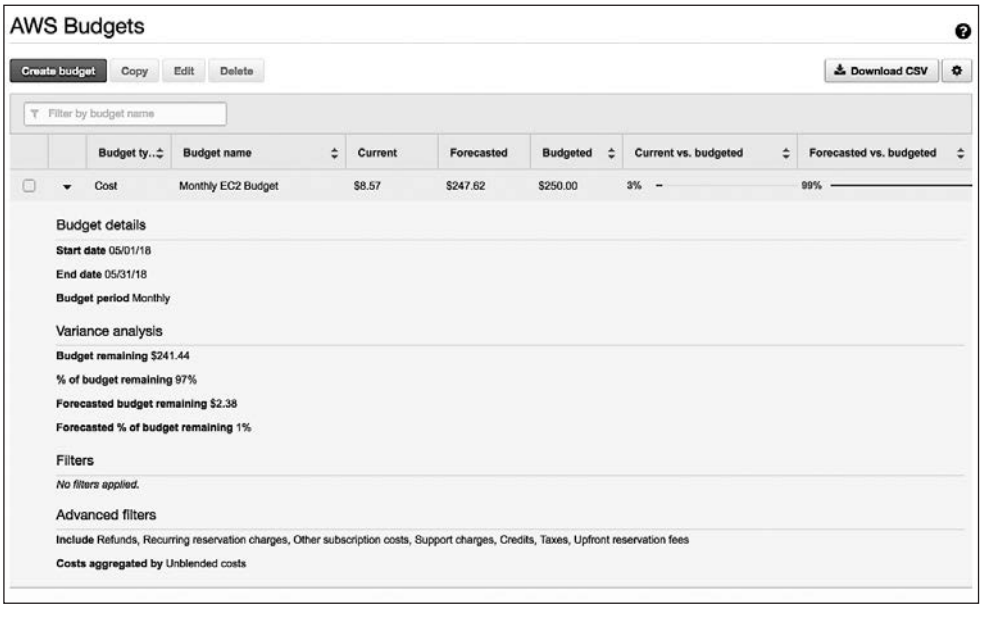


Рис. 7.2

Установка бюджетов — это эффективный способ мониторинга затрат на облачные вычисления. Системы автоматически уведомят вас, когда вы приблизитесь к установленной сумме расходов или превысите ее (рис. 7.3).

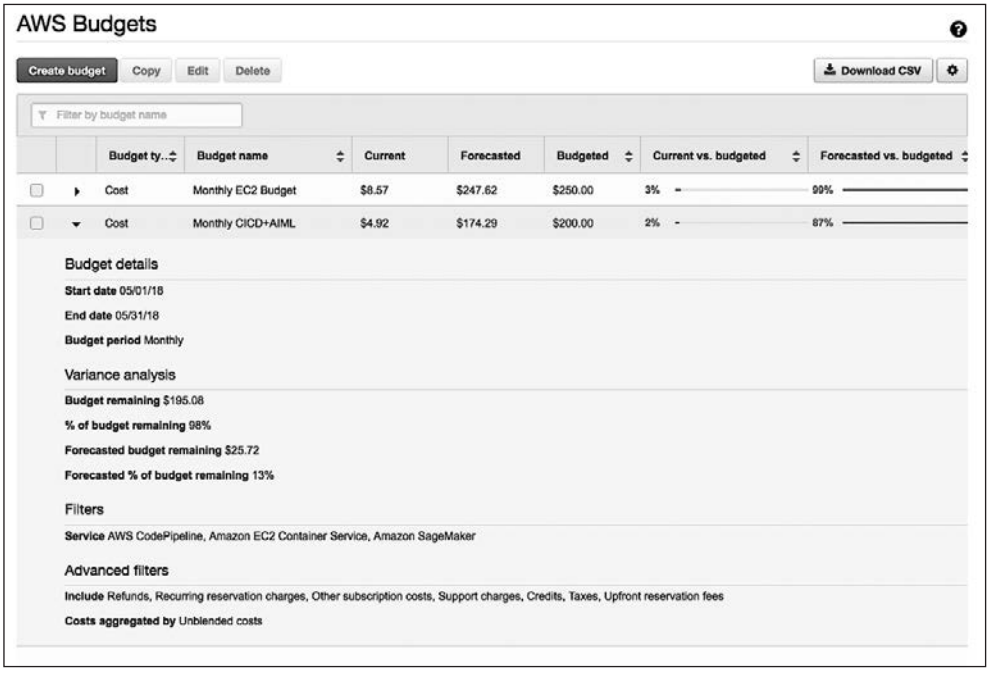
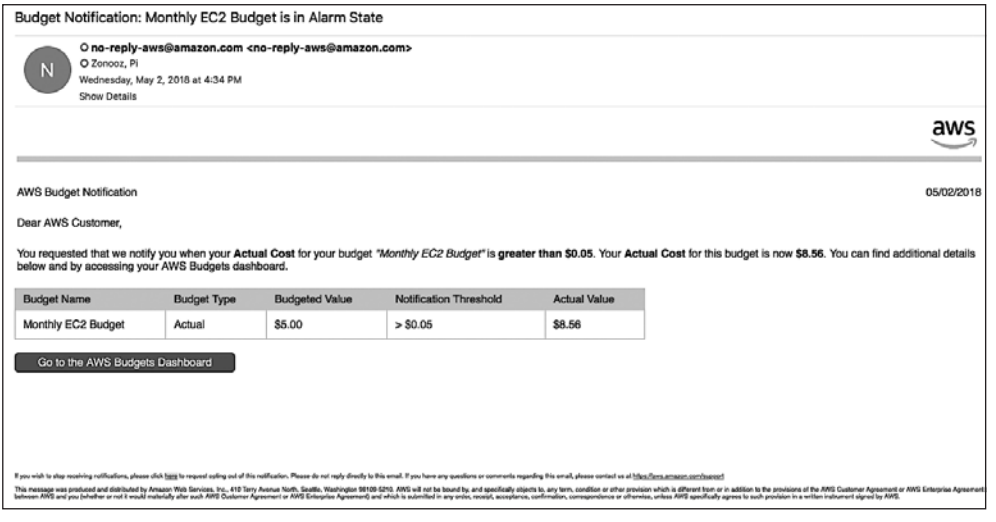


Рис. 7.3

Оповещения о выставлении счетов можно настроить на отправку уведомлений по электронной почте (рис. 7.4). Это позволяет минимизировать перерасход и оптимизировать затраты на облачную среду.

Существуют также работающие на различных облачных платформах сторонние инструменты для агрегирования, отображения и прогнозирования затрат (они будут описаны в конце главы).

Неизменяемые архитектуры требуют уникального подхода, учитывающего передовой опыт развертывания и управления, рассмотренный в предыдущих главах. Поскольку каждое развертывание реплицирует весь стек или его модульный компонент, можно оценить его перед развертыванием. Работая с AWS Cloudformation, на последней странице подтверждения перед развертыванием вы можете согласиться с ежемесячной стоимостью стека, используя ссылку Cost рядом с Estimate Cost. Это перемещает вас на страницу с уже заполненным простым калькулятором, который подсчитывает ежемесячную стоимость (рис. 7.5).


Budget Notification: Monthly EC2 Budget is in Alarm State

no-reply-aws@amazon.com <no-reply-aws@amazon.com>
Zonooz, Pi
Wednesday, May 2, 2018 at 4:34 PM
Show Details

aws

AWS Budget Notification 05/02/2018

Dear AWS Customer,

You requested that we notify you when your **Actual Cost** for your budget *"Monthly EC2 Budget"* is **greater than $0.05**. Your **Actual Cost** for this budget is now **$8.56**. You can find additional details below and by accessing your AWS Budgets dashboard.

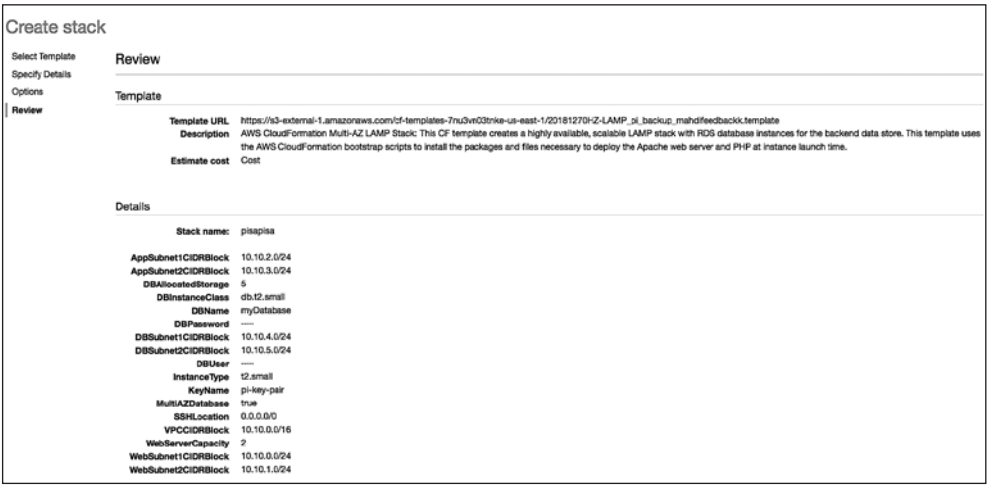
Budget Name	Budget Type	Budgeted Value	Notification Threshold	Actual Value
Monthly EC2 Budget	Actual	$5.00	> $0.05	$8.56

Go to the AWS Budgets Dashboard

If you wish to stop receiving notifications, please click here to request opting out of this notification. Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

This message was produced and distributed by Amazon Web Services, Inc., 410 Terry Avenue North, Seattle, Washington 98109-5210. AWS will not be bound by, and specifically objects to, any term, condition or other provision which is different from or in addition to the provisions of the AWS Customer Agreement or AWS Enterprise Agreement between AWS and you (whether or not it would materially alter such AWS Customer Agreement or AWS Enterprise Agreement) and which is submitted in any order, receipt, acceptance, confirmation, correspondence or otherwise, unless AWS specifically agrees to such provision in a written instrument signed by AWS.

Рис. 7.4


Create stack

Select Template
Specify Details
Options
Review

Review

Template

Template URL https://s3-external-1.amazonaws.com/cf-templates-7nu3vn03tnke-us-east-1/20181270lrZ-LAMP_pi_backup_mahdifeedbackk.template
Description AWS CloudFormation Multi-AZ LAMP Stack: This CF template creates a highly available, scalable LAMP stack with RDS database instances for the backend data store. This template uses the AWS CloudFormation bootstrap scripts to install the packages and files necessary to deploy the Apache web server and PHP at instance launch time.
Estimate cost Cost

Details

Stack name:	pisapisa
AppSubnet1CIDRBlock	10.10.2.0/24
AppSubnet2CIDRBlock	10.10.3.0/24
DBAllocatedStorage	5
DBInstanceClass	db.t2.small
DBName	myDatabase
DBPassword	----
DBSubnet1CIDRBlock	10.10.4.0/24
DBSubnet2CIDRBlock	10.10.5.0/24
DBUser	----
InstanceType	t2.small
KeyName	pi-key-pair
MultiAZDatabase	true
SSHLocation	0.0.0.0/0
VPCCIDRBlock	10.10.0.0/16
WebServerCapacity	2
WebSubnet1CIDRBlock	10.10.0.0/24
WebSubnet2CIDRBlock	10.10.1.0/24

Рис. 7.5

При управлении развертыванием в виде стека, выраженного в коде, каждый стек можно оценить до развертывания (как в случае развертывания стека CloudFormation на AWS).

На рис. 7.6 показан пример сгенерированной калькулятором ежемесячной сметы, которая была создана из образца развертывания шаблона CloudFormation.

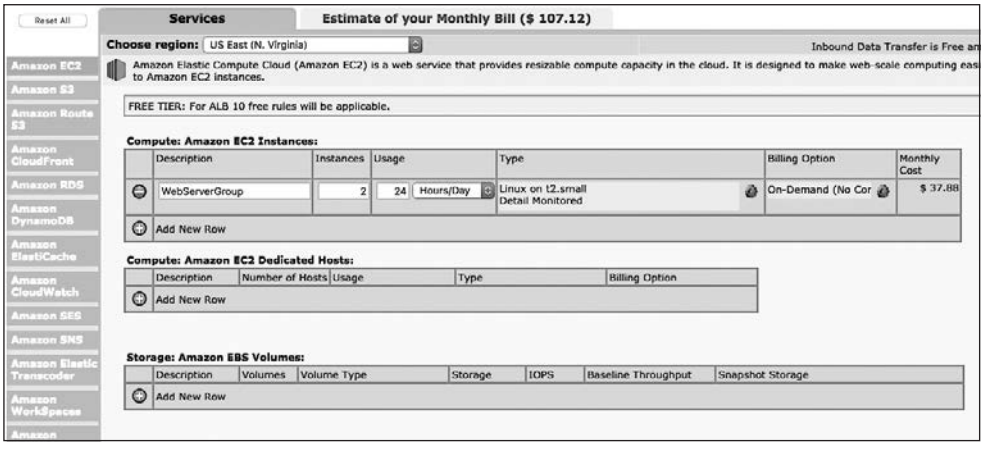


Рис. 7.6

Рекомендации по использованию тегов

Как для крупных предприятий, так и для небольших стартапов тегирование является наиболее важным ежедневным действием, которое необходимо выполнять для обеспечения прозрачности затрат. CSP могут прозрачно отчитываться и отображать биллинг, но эти сведения мало что значат для бизнеса или конечных пользователей, если они не могут распределять или отслеживать затраты по ключевым бизнес-функциям.

Тегирование — это нативная функция, поддерживаемая всеми ведущими облачными провайдерами. Поскольку у любых предприятия, организации и потребителя есть уникальные внутренние процессы и терминология, важно разработать стратегию тегирования, которая подойдет именно вам.

Практические рекомендации для архитектуры cloud native

Тегирование очень быстро станет обременительным и его станет невозможно поддерживать, если теги не будут обозначены во время запуска. В течение пары недель вся среда может быть немаркирована, и, следовательно, ею невозможно будет управлять, если жизненные циклы разработки довольно короткие. Облачная среда с оптимизацией затрат автоматически обнаруживает и даже удаляет непомеченные ресурсы. Это очень быстро учит команды относиться к тегированию как к критически важной деятельности.

Задать автоматическое применение тегов можно несколькими способами. С помощью интерфейса командной строки можно создать список нетегированных ресурсов для каждого сервиса (например, AWS EC2, EBS и т. д.). Встроенные инструменты

CSP, такие как AWS Tag Editor, можно использовать для поиска нетегированных ресурсов вручную. Оптимальным облачно-ориентированным подходом было бы создание правила, требующего тегирования для облачных сервисов, которые делают это автоматически, например AWS Config Rules. Правила конфигурации постоянно проверяют среду на наличие тегов, которые вы указываете по необходимости. Если обнаружится, что этого не происходит, правила можно откорректировать вручную или автоматически. На рис. 7.7 показано, как правила AWS Config позволяют автоматически обнаруживать ресурсы, не помеченные тегами, и составлять отчеты о них.

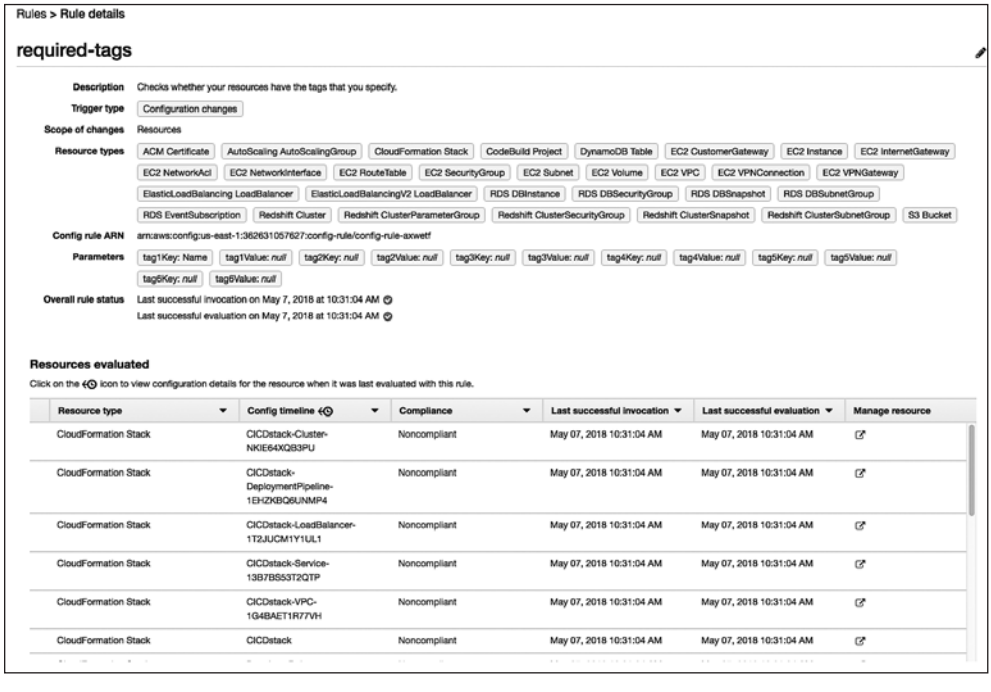


Рис. 7.7

На рис. 7.8 изображено, как с помощью редактора тегов AWS можно вручную искать ресурсы без маркировки, хотя это более громоздкий процесс, чем автоматическое обнаружение, особенно в крупных корпоративных средах, где используются и потребляются тысячи ресурсов.

В развитой среде каждым развертыванием управляют как кодом с помощью конвейеров развертывания. В этом конвейере следует использовать шлюз, чтобы обеспечить использование подходящих тегов в шаблоне. Если все сделано правильно, во время развертывания теги должны быть автоматически назначены всем ресурсам в шаблоне. Ключевым аспектом назначения тегов является автоматизация, и, назначая теги в конструкции верхнего уровня (шаблон стека), мы минимизируем необходимость ручного вмешательства и повышаем точность тегов.

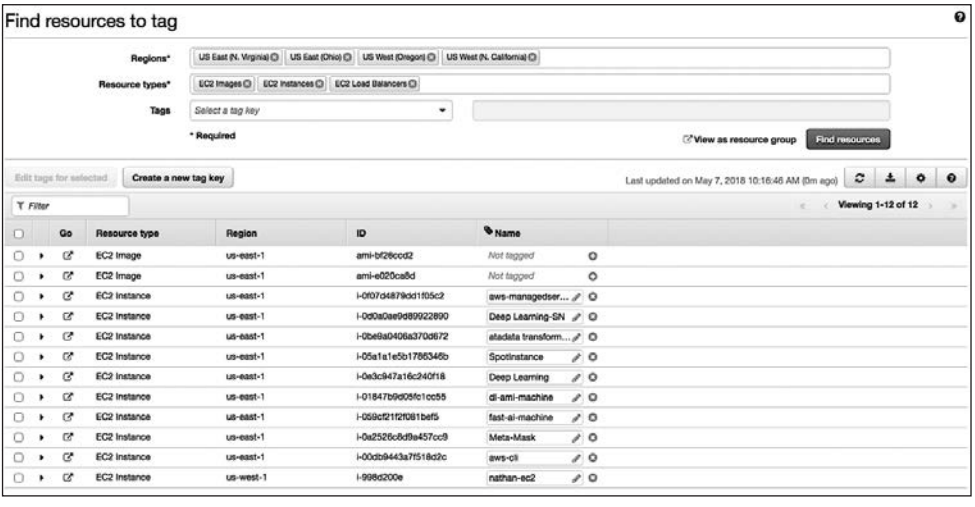


Рис. 7.8

Сокращение затрат

Теперь, когда у нас есть подход для сбора, отслеживания и просмотра затрат в облачной среде, как оптимизировать эти затраты? Существует два лагеря, придерживающихся разных точек зрения по поводу *оптимизации* затрат. Один лагерь слышит: *минимизируй* (чтобы уменьшить до минимально возможного уровня), а другой — *максимизируй* (чтобы использовать наилучшим образом). Оба подхода технически верны, поскольку ни у одной ИТ-организации нет бездонного бюджета (следовательно, вы минимизируете расходы в рамках своего бюджета). Однако ведущие организации изменили свой взгляд на ИТ-расходы. Они стремятся получить максимальную выгоду для бизнеса и увязать инвестиции в технологии с бизнес-результатами. Это то, что подразумевается под *максимизацией/оптимизацией* затрат на технологии. Первая требует технологических решений, а вторая — технических экспериментов, терпения и деловой хватки.

Оптимизация вычислений

Обычно наибольшая экономия в облачной среде достигается при правильном выборе размера вычислительных ресурсов. Поскольку наибольшая доля затрат обычно приходится на вычисления, разумно ожидать, что здесь же можно и максимально сэкономить. *Правильный выбор размера* означает выбор виртуальной машины, лучше всего подходящей для данного приложения (размер здесь означает количество процессоров и памяти, выделенных машине).

Эта проблема особенно остро проявляется в средах, которые только что были перенесены из локальной в облачную среду. Процесс определения размера ресурсов для

локальной среды обычно требует множества согласований в разных группах, каждая из которых усиливает защиту CYA. Это приводит к появлению чрезмерно мощных машин, которые затем переносятся на сопоставимые виртуальные машины в облаке.

Практические рекомендации для архитектуры cloud native

После успешных миграции или развертывания облачного приложения определите базовый уровень применения для каждого из ваших вычислительных ресурсов. Используйте эти данные, чтобы правильно настроить нужный уровень производительности виртуальных машин.

Если ваше приложение может работать в кластерах и не имеет состояния, стоит использовать больше инстансов меньшего размера. Это наилучшим образом оптимизирует затраты (и производительность), поскольку вы сможете масштабировать кластер, чтобы улучшить соответствие спросу. AWS и Azure предлагают широкий выбор виртуальных машин, которые подходят для любого варианта использования. GCP предлагает специальные типы машин, где пользователь может установить доступные размеры. Это подразумевает еще больший контроль над соответствием физических ресурсов требованиям приложения.

Оптимизация хранилища

В счетах за облачные услуги затраты на хранение обычно занимают второе место после затрат на вычисления. Часть этих затрат идут на виртуальные диски/тома, подключенные к каждой виртуальной машине; сервисы хранения объектов, в которых размещаются ресурсы приложений, изображения/видео и документы; использование сервиса архивных данных; резервное копирование этих данных в сервисе резервного копирования.

Для всех форм хранения данных важно установить максимально автоматизированный жизненный цикл. Данные и резервные копии должны естественным образом продвигаться вниз по этапам этого жизненного цикла, пока не будут помещены в архивы в соответствии с требованиями или удалены. С помощью тегирования следует обозначать, какие хранилища данных или объекты необходимы в долгосрочной перспективе для соблюдения соответствия. Для управления ими следует задействовать сценарии, хотя существуют облачные сервисы, помогающие в этом пользователям.

Simple Storage Service (S3) от AWS — это надежный сервис для хранения объектов. S3 поддерживает политики жизненного цикла, которые позволяют переходить от стандарта S3 к более дешевым и менее производительным сервисам, например, Standard-IA (нечастый доступ), One Zone-IA и Glacier (долгосрочное архивное холодное хранилище). Все три сервиса перехода дешевле, чем стандартный сервис S3. Можно также настроить окончательное удаление объектов через *X* дней после создания. Это можно установить на уровне корзины, которая работает со всеми папками и объектами.

Диспетчер жизненного цикла данных для AWS EBS также предоставляет собственное решение для управления сотнями или тысячами резервных копий на уровне томов. Для крупных предприятий эта задача окажется обременительной, поскольку необходимо будет создавать автоматизированные инструменты для запроса, сбора информации и удаления старых или просроченных снапшотов. Этот сервис от AWS предоставляет пользователям стандартные средства для организации этих жизненных циклов.

Результаты применения бессерверного подхода

Бессерверные архитектуры представляют собой уникальное решение с точки зрения затрат. Большая часть результативных усилий по оптимизации затрат приходится на написание эффективного кода, призванного сократить время выполнения или количество случаев выполнения. Это очевидно из модели ценообразования на сервисы выполнения бессерверного кода, такие как AWS Lambda, который взимает плату в зависимости от количества случаев выполнения, времени выполнения и выделенной памяти для контейнера, в котором работает код (https://s3.amazonaws.com/lambda-tools/pricing-calculator.html). Размер памяти можно оптимизировать, отслеживая объем памяти, используемой при выполнении (это можно делать в AWS CloudWatch).

Другие *бессерверные* облачные сервисы, такие как AWS Kinesis и Athena, задействуют аналогичную модель ценообразования на основе данных (за час или на единицу полезной нагрузки для Kinesis, за 1 Тбайт данных, отсканированных для Athena). Эти сервисы почти всегда дешевле, чем сопоставимые, такие как Apache Kafka и Presto, которые размещаются на самоуправляемых вычислительных узлах.

Облачный инструментарий

Многие облачные инструменты для оптимизации затрат доступны на соответствующих платформах. Однако два фактора (справедливо) могут подтолкнуть компании к использованию внешних сервисов для оптимизации затрат.

- Независимость от третьей стороны в определении областей оптимизации затрат.
- Возможность работы в двух или более облачных средах (мультиоблачные архитектуры).

Конкуренция в этой области усиливается, поскольку облачные провайдеры копируют многие функции, предоставляемые этими компаниями, на своих платформах. Многие инструменты, обеспечивающие оптимизацию затрат, были вынуждены прибегнуть к автоматизации управления и эксплуатации.

Cloudability

Один из последних независимых инструментов, ориентированных на прозрачность затрат, — Cloudability, который предоставляет пользователям подробный бюджет и ежедневные отчеты на основе множества показателей, которые он собирает из ваших облачных сред. В настоящее время он доступен и интегрируется с основными облачными провайдерами.

AWS Trusted Advisor

Этот сервис от AWS еще будет упоминаться в нескольких главах книги из-за его ценности для облачных пользователей AWS. Trusted Advisor дает рекомендации по автоматической оптимизации затрат, рискам безопасности, оптимизации производительности и отказоустойчивости. Trusted Advisor проводит множество проверок для минимизации затрат, уведомляя пользователей, к примеру, о выборе инстанса неподходящего размера, простаивании ресурсов и т. д.

Azure Cost Management

Благодаря приобретению Cloudyn инструмент Azure Cost Management обеспечивает эффективный способ отслеживания использования ресурсов и расходов в вашей среде Azure. Он дает рекомендации по правильному выбору размера инстанса и определяет неиспользуемые ресурсы.

Резюме

В данной главе мы рассмотрели структуру затрат современных предприятий, управляющих собственными центрами обработки данных. Мы определили общую терминологию, чтобы нам было легче понимать и оценивать рентабельность внедрения облачной инфраструктуры.

Мы узнали о том, как облачные решения позволяют своим клиентам существенно экономить и пользоваться огромными преимуществами, а также о том, как модель ценообразования сместилась с капитальных затрат на операционные, что является ключевым отличием облачных систем от устаревших.

Наконец, мы разработали стратегии тегирования, которые помогут отследить затраты и подробно их проанализировать. Эти стратегии помогают оптимизировать архитектуры.

В следующей главе поговорим об оптимизации в сфере эксплуатации.

8 Эксплуатация облачных сервисов

В предыдущих главах мы узнали, что делает облачную архитектуру уникальной и отличающейся от прежних моделей построения ИТ-систем. На наш взгляд, облачные технологии не ограничиваются только архитектурными решениями основных технологических компонентов данной системы. Чтобы извлечь максимальную выгоду, параллельно с внедрением облачных технологий должны развиваться методы эксплуатации. Без ориентированности организации на облачные технологии большинство преимуществ облака будут не реализованы.

В этой главе вы узнаете, как создать эффективную организацию, способную проектировать, разрабатывать и поддерживать облачную среду, которая представляет собой сочетание процессов, специалистов и инструментов, объединенных для реализации всего потенциала *облачных разработок (CND)*.

К концу этой главы вы сможете получить ответы на следующие вопросы.

- Как облако повлияло на обязанности среднего ИТ-специалиста?
- Как выглядит успешная облачная организация?
- Как создать механизмы, обеспечивающие применение лучших облачных практик?
- Как выглядят основные инструменты и процессы?
- Как создать организацию, которая усиливает возможности разработчиков?
- Что такое облачная культура и почему она важна?

Прежде чем перейти к облаку

Как всегда, будет лучше, если мы начнем издалека. Вне зависимости от того, переводите ли вы свою ИТ-среду и организацию на использование облачных технологий, или вам еще только предстоит начать этот путь, вы должны понимать, как организовано большинство компаний в плане проектирования, разработки и эксплуатации современных ИТ-систем.

В качестве примера рассмотрим страховую компанию, которая предлагает своим клиентам несколько продуктов — страхование жилья, жизни и автомобиля. В компании есть несколько команд, которые помогают поддерживать ее продукты: разработчики политик, группы продаж, работающие непосредственно с клиентами, статистики, HR-специалисты, программисты, маркетологи, секретари и т. д. Каждая из этих команд пользуется услугами ИТ-отдела, поддерживающего работоспособность серверов для электронной почты, кластеров для запуска статистических моделей, баз данных для хранения информации о клиентах и сайтов, которые служат витриной для потенциальных клиентов.

ИТ-отдел состоит из нескольких команд в зависимости от предметной области. Сеть, безопасность, база данных и эксплуатация — это области, за которые отвечают типичные команды, встречающиеся в таких устаревших организационных моделях. Эти команды работают параллельно и отдельно от команд разработчиков, которые полностью сосредоточены на написании кода (рис. 8.1).

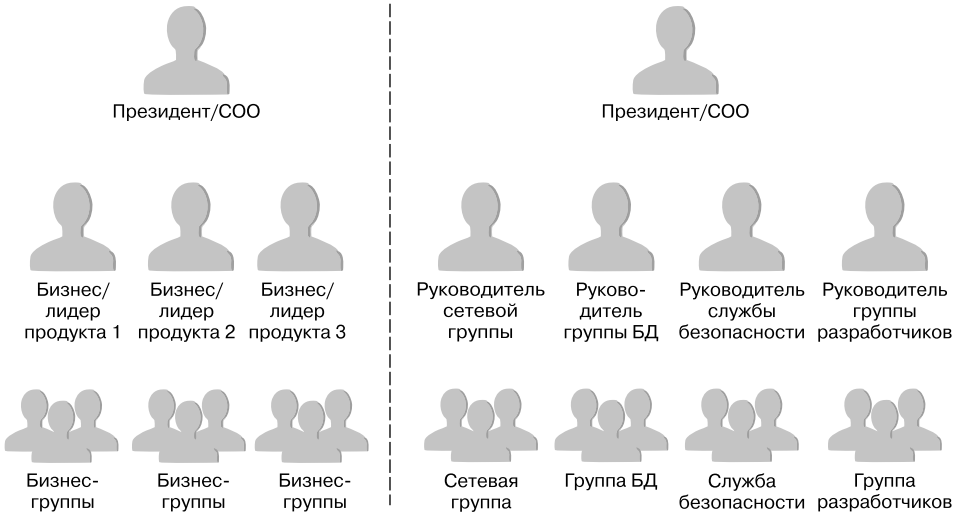


Рис. 8.1

На схеме показана организационная структура типичной компании, разрабатывающей продукт. Существует строгое разделение между экспертами в области технологий и владельцами продуктов/сервисов. Проблема здесь в том, что, по сути, эти организации не сотрудничают друг с другом. Генеральному директору все видится так, будто те, кто находится в левой части схемы, пытаются максимизировать доходы, открыть новые рынки и привлечь больше клиентов. Правую часть схемы типичный генеральный директор рассматривает как центр затрат — группу, которую необходимо минимизировать и оптимизировать до бесконечности.

Такое отношение отражается в поведении людей, корпоративной культуре, психологии лидеров и усилиях, предпринимаемых для того, чтобы наладить сотрудничество между этими двумя группами. ИТ-команды рассматривают бизнес-группы как иностранные организации, выдвигающие нереалистичные требования к созданию систем в слишком короткие сроки, с нереалистичными функциями, к тому же за очень небольшую плату. А бизнес-группы считают, что деятельность ИТ-команд не связана с основной целью организации и проблемами клиентов, которые они пытаются решить. ИТ-специалисты не стремятся понять цели бизнеса и, кажется, негативно реагируют на каждый запрос.

Среди ИТ-специалистов ситуация только усугубляется, поскольку в каждой предметной области команды состоят из единомышленников. Специалисты по безопасности работают в основном с другими специалистами по безопасности и сидят вместе с ними. То же самое верно для тех, кто занят сетями, БД, эксплуатацией и архитектурой. Если взглянуть на эту модель сверху вниз, она способствует возникновению условий, в которых одни и те же идеи снова и снова высказываются разными людьми, не встречая сопротивления и не проходя должной проверки со стороны тех, кто не специализируется в этой области. В данной модели предметные подходы становятся негибкими и хрупкими. Любая проблема, связанная с тем, *как мы работаем с чем-либо*, встречает строгий запрет или длительную процедуру согласования с участием высшего руководства. Такой ограниченный доступ к ресурсам вреден для отдельных работников. Они отрезаны от остальной части предприятия и технических команд, поэтому возможность учиться и расширять свои знания в целом для них ограничена.

Все руководители команд (безопасность, сеть и т. д.) будут защищать свои ресурсы и расширять сферы влияния, чтобы закрепить команды группы в организации. Они стремятся установить ограничения, регламентировать процессы взаимодействия членов своей команды и других отделов. Они устанавливают правила, когда и как другие команды могут привлекать их сотрудников, и диктуют, в каких областях требуется их одобрение. Они не только пытаются контролировать скорость, с которой члены команды могут работать, но и ограничивают способность других команд быстро и беспрепятственно развиваться.

Для проектов, которые требуют сочетания ресурсов нескольких технологических команд, существует строгий процесс проверки, когда руководители должны провести анализ, обсудить, взять на себя обязательства и обеспечить контроль, прежде чем можно будет собрать смешанную команду. Часто такие команды все так же сталкиваются с множеством препятствий, о которых говорилось ранее. Их деятельность, как правило, сводится к структуре отчетности, в рамках которой принятие любого решения по-прежнему должно согласовываться с руководителями команды. Данные структуры являются противоположностью независимой автономной команды, которая может принимать решения и итеративно развивать продукт.

Апофеозом такого подхода к организации проектов является *совет по утверждению изменений (Change Approval Board, CAB)*. Он возник из-за необходимости контролировать любые изменения в монолитной системе, где они могут иметь множество последствий. Незначительная корректировка конфигурации может создать каскад проблем в организационной структуре, вывести из строя системы, приносящие доход, и вызвать десятки событий с уровнем серьезности 1.

CAB был создан по необходимости в то время, когда системы работали на общих аппаратных ресурсах, сервисы внутри системы были тесно связаны и производительность системы в целом (кроме скорости вывода) была непрозрачна. По мере развития технологий и системной архитектуры CAB превратился в пережиток прошлого. Такие советы по-прежнему есть во многих ИТ-организациях, несмотря на технологические ограничения, из-за которых CAB в своем изначальном виде больше не существует.

Наряду с CAB возникли *база данных управления изменениями (CMDB)* и база данных управления конфигурацией. Назначение этих систем:

- контроль и отслеживание запланированных изменений, внесенных в монолитную систему;
- отслеживание текущего состояния разрастающейся системы.

Возникновение этих систем было продиктовано необходимостью, но в мире современных облаков обе они являются устаревшими. Это связано с процессами, производимыми вручную и необходимыми для поддержания актуальности информации, хранящейся в CMDB. Практичность обслуживания этих систем недостаточна, к тому же часто они хранят очень мало информации более высокого порядка, делающей такое хранилище данных полезным для пользователя. Сохраненные данные необходимо регулярно обновлять и поддерживать, в противном случае они устареют и потеряют всякую актуальность. Таким образом, организации приходится прикладывать значительные усилия к обслуживанию этой БД, на что затрачивается часть времени и ресурсов, необходимых для разработки и создания продуктов. Методы, пришедшие на смену CMDB, будут описаны позже в этой главе, а сейчас достаточно сказать, что в них делается больший упор на автоматизацию, постоянное хранение и актуальность данных.

Важнейшим средством организации и развития проектов наряду с CAB и CMDB стала электронная почта. Ею пользуются все команды в отдельно взятой предметной области, чтобы документировать процесс принятия решений. Команды применяют электронную почту для всего. Простые решения, которые можно было бы принять в ходе пятиминутной беседы лицом к лицу, принимаются по электронной почте. Электронная почта создает важную контекстуальную историю, помогающую понять, как мы пришли к решению, но не позволяет командам быстро получать

информацию. То, что можно обсудить за пять минут при личной беседе, может потребовать переписки в течение дня, а то и дольше в зависимости от того, как быстро члены команды отвечают на электронные письма.

Развитие облачных технологий

В предыдущих главах мы представляли монолит с помощью разделенных легкодоступных архитектур, состоящих из микросервисов. Эти сервисы были созданы таким образом, что можно было управлять ими и развивать их независимо от других компонентов системы. Технологические сервисы, лежащие в основе этих микросервисов, быстро совершенствовались, но организации, создававшие системы, оставались монолитными.

Благодаря кропотливым упорным экспериментам такие компании, как Amazon, обнаружили, что лучший способ создания систем, основанных на разделенных микросервисах, — создание микрокоманд специалистов. Эти команды олицетворяют микросервисные архитектуры — небольшие, несвязанные и автономные. В Amazon их называют *командами двух пицц*, поскольку ввиду небольшого размера команды ее членов можно накормить всего двумя пиццами. Мы будем называть их CND-командами.

Участники каждой CND-команды обладают необходимыми талантами и навыками, нужными для разработки, создания, предоставления и эксплуатации сервисов. Они представляют собой автономную организационную единицу, которая может управлять всем жизненным циклом конкретного микросервиса. Им не нужно получать одобрение других владельцев продуктов или вице-президентов, чтобы решить, какой конкретный набор инструментов или фреймворков приемлем или могут ли они внедрить новое изменение в производство.

Они руководствуются двумя главными принципами.

- Все должно управляться API, и как только API публикуется для внешних сервисов, необходимо приложить максимум усилий, чтобы никогда не изменять его, — *API навсегда*.
- Каждая команда занимается своим продуктом от разработки до запуска. Команды заинтересованы в создании стабильных продуктов, которые работают при минимальном участии человека.

Эти два принципа служат естественными стимулами для того, чтобы команды разрабатывали стабильные микросервисы, которые хорошо сочетаются с другими в системном ландшафте. Команды не могут перебрасывать ошибочный или грозящий ошибками код группам эксплуатации, потому что они и *есть* группа эксплуатации (это корень термина DevOps). Члены команды должны по очереди дежурить в соответствии с графиком, который они все согласовали, и отвечать

за плохо выполненные сборку или проект. Поскольку члены команды регулярно не спят ночами, исправляя неполадки с уровнем серьезности 1 или изменяя архитектуру систем, в дальнейшем они будут стремиться создавать в своих системах архитектуру высокой доступности и не склонную к хрупкости.

Это в корне меняет команды, превращая их из однородных групп людей с одинаковой квалификацией в группы специалистов, имеющих любые навыки, необходимые для выполнения работы. Команды развивают многопрофильные навыки и воплощают свои амбиции, нанимая талантливых работников. Кроме того, все члены команды совершенствуют рабочие умения, которые используют для создания более надежных систем. Команды CND часто состоят из разработчиков внешнего интерфейса и серверной составляющей, специалистов по базам данных и безопасности, а также экспертов по эксплуатации, которые помогают друг другу создавать более стабильный сервис.

Поскольку каждая группа CND, управляющая микросервисом, небольшая, количество каналов связи между специалистами становится управляемым. Они могут принимать решения очень быстро, не ориентируясь на все то, что этому предшествовало. Это означает, что электронную почту можно отставить, а мессенджеры (например, Slack) использовать для постоянного отслеживания обсуждений, в которых участвуют члены команды. С помощью чата специалисты могут быстро общаться и договариваться. Отдельные каналы #channels применяются для разделения беседы на комнаты, относящиеся к разным доменам, связанным с командой.

CND-команды сосредотачиваются на создании повторяемых процессов, которые помогают им как можно быстрее перейти от проектирования к развертыванию. Это означает создание конвейеров, автоматизирующих многие механические задачи, выполняемые вручную не облачными командами. Мы рассмотрим инструменты и стратегию позже в этой главе. Процесс должен быть адаптирован к максимальному количеству небольших развертываний в производственной системе.

В облачном мире методы CMDB, предназначенные для мониторинга довольно статичного физического оборудования, обречены на провал. Управление изменениями и конфигурациями должно происходить более динамично, поскольку сотни изменений могут произойти в течение часа. В данном случае следует ориентироваться на сервис непрерывного обнаружения, основанный на следующих факторах.

- Инфраструктура как код и образы машин.
- Репозитории кода, хранящие исходный код.
- Конвейеры с проверкой результатов на каждом этапе (при необходимости вручную) для ускорения развертывания.
- Приложения для группового чата с интеграцией API, которые могут уведомлять команды об изменениях, запрашивать согласие на них или отказ.

- Облачные сервисы конфигурации, такие как AWS Config, которые показывают историю конфигурации и позволяют разрабатывать правила для оценки соответствия.

Управление облачными приложениями также становится более динамичным. Комбинируя такие сервисы, как AWS Config и автоматические триггеры кода по событиям, можно разработать механизмы для автоматического обнаружения и применения конфигураций, а также для создания соответствующих отчетов.

Команды разработки, ориентированные на облако

Чтобы в полной мере использовать преимущества облачных технологий в качестве инструмента поддержки бизнеса, команды должны быть реорганизованы в автономные организации, которые мы называем CND-командами. Принципы CND следующие.

- Полное владение своим сервисом.
- Оптимизация процесса принятия решений в группе (быстрое достижение консенсуса).
- Ротация ответственных за сопровождение (каждый дежурит по очереди).
- Постоянные публикация и поддержка API.
- API — это единственный способ взаимодействовать с другими сервисами.
- Правильные инструменты — это те, которые выполняют свою работу (команда определяет, какие язык, фреймворк, платформа, движок и т. д. подходят для их обслуживания).
- Автоматизация важнее, чем новые функции (в долгосрочной перспективе автоматизация позволяет быстрее реализовать больше функций).

Чтобы максимально увеличить скорость разработки, команды должны полностью реализовывать процесс проектирования и принятия решений. Принимая решения, они не могут полагаться на отдел кадров или высшее руководство, так как это сильно замедлит процесс, поскольку придется в подробностях объяснять руководителю, почему это решение необходимо.

Члены команд могут решать в своем кругу, как они хотят упростить процесс принятия решений и какие проектные решения требуют более широкого совместного обсуждения. Они могут выбирать собственные способы достижения консенсуса, исходя из сильных и слабых сторон специалистов. Кроме того, они могут решать, как будет осуществляться оперативный надзор за их сервисами. Тут и потребуется механизм достижения консенсуса, чтобы договориться о том, кто и когда *дежурит*.

Можно сомневаться в том, насколько эффективными будут CND-команды, когда каждый сервис станет зависеть от других сервисов или систем, управляемых другими командами. Это не проблема, если соблюдается золотое правило: любое взаимодействие с внешними сервисами должно осуществляться через API. API публикуются и поддерживаются вечно. Это гарантирует, что любые системы, разработанные по отдельности, но интегрирующиеся в основной сервис, могут полагаться на устойчивый, неизменяемый программный интерфейс. Однако это не означает, что первичная система не способна изменяться. Фактически изменение может происходить быстрее, поскольку можно менять местами базовые системы (например, механизмы СУБД или фреймворки), но программный интерфейс (связующее звено, объединяющее разрозненные сервисы) останется неизменяемым. Это правило работает по необходимости, поскольку изменение API требует, чтобы команда поняла, какими будут результаты этого изменения. Если бы команде пришлось фиксировать все возможные изменения и оценивать их влияние, ее участники не смогли бы полностью абстрагироваться от других сервисов.

Команды двух пицц

Основная проблема крупных организаций в том, чтобы достичь консенсуса. Это необходимо для продвижения большинства проектов, так как разработка технологических систем носит взаимозависимый характер. Поскольку системы, как правило, большие и над ними работает несколько команд, внесение одной командой изменений в свою часть системы может вызвать неожиданные последствия для подсистем, принадлежащих другим командам и поддерживаемых ими. Вот несколько примеров.

- *Эффект восходящего потока*. Обновленная подсистема работает так быстро, что перегружает поток данных запросами новых данных.
- *Эффект нисходящего потока*. Обновленная подсистема генерирует данные намного быстрее, и нижестоящий потребитель не успевает их обрабатывать.

При широком внедрении разделенных систем и SOA мы решаем эту проблему с технологической точки зрения, но организации еще предстоит адаптироваться к новой реальности. Если у нас есть разделенные системы, почему бы не разделить организационные единицы? Так и появилась команда двух пицц. В разных организациях ее называют по-разному: команда тигров, команда DevOps, команда запуска сборки, спецназ и т. д. Мы же называем ее CND-командой.

Такие команды создают, запускают и обслуживают свои системы, но почему их называют командами двух пицц? Их можно накормить максимум двумя пиццами, а это значит, что состоят они не более чем из 8–12 человек. Откуда взялся этот лимит? Принять решение и достичь консенсуса в большой группе трудно.

Это вызвано увеличением количества каналов связи между *n*-м количеством разработчиков. С увеличением *n* число связей внутри группы путей увеличивается экспоненциально (рис. 8.2).

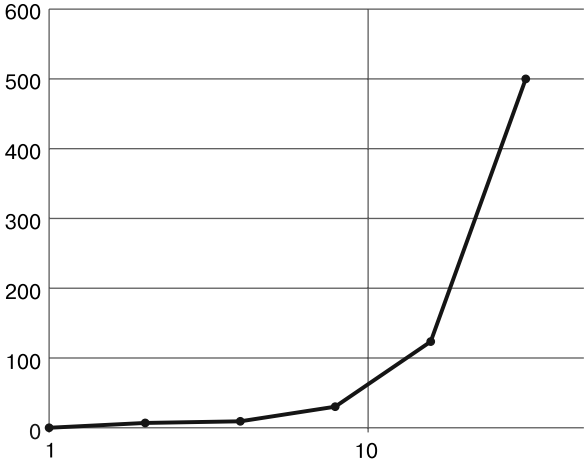


Рис. 8.2

Когда в команде более десяти человек, количество связей между ними увеличивается в геометрической прогрессии. Эта тенденция присутствует с самого начала, но становится гораздо заметнее. И наоборот, *небольшие команды могут быстрее достичь консенсуса*. В этом причина ограничения их численности.

Поставщики сервисов с облачным управлением

Корпоративные организации часто сотрудничают с поставщиками управляемых сервисов (Managed Service Providers, MSP), чтобы иметь возможность оперативно управлять своими облачными средами. Уже довольно давно наблюдается ситуация, когда провайдеры CSP берут на себя *тяжелые единообразные задачи* (которые изначально заключались в управлении физическим оборудованием), но все никак не могут продвинуться вверх по стеку, где происходит администрирование кластеров виртуальных машин для контейнеров или баз данных, развернутых с расчетом на высокую доступность.

Эти MSP предлагают управление средой наряду с широким спектром услуг, в которых повышенная ответственность возлагается на клиента, с одной стороны, и почти все управление средой на MSP — с другой. MSP обычно предлагают эти услуги за счет сокращения количества инструментов и ограничения выбора платформ для

разработки. Для MSP может потребоваться конкретный конвейер развертывания. Они могут принудительно использовать инструменты оркестрации контейнеров, такие как Kubernetes поверх Mesosphere, принудительно отправлять все запросы на изменение через такие инструменты, как Zendesk, а также управлять идентификацией и доступом с помощью Okta.

Хотя MSP могут помочь рационализировать сложную инструментальную среду и снизить операционную нагрузку, мы считаем, что они являются ступенькой к истинно облачной архитектуре в соответствии с *моделью зрелости облачной среды (Cloud Native Maturity Model, CNMM*).

По сути, MSP представляют собой отказ тех, кто занимается сопровождением системы, от ответственности за ее развитие и управление изменениями. Не размышляя о процессе изменения, а также о его цели или характере, создать облачную архитектуру может быть очень сложно.

Продвинутые организации, применяющие третий принцип модели CNMM, автоматизацию, имеют возможность создавать и развивать свои системы облачно-ориентированым образом, и именно эти системы порождают облачные архитектуры. Невозможно просто достичь зрелости модели. Это процесс, который требует организационных и культурных изменений для построения систем такого типа и управления ими.

Работа с IaC

Процесс администрирования и развертывания кода для IaC похож на то, как разработчики управляют кодом своих приложений.

Изначально CND-команды хранят свой IaC-код в репозиториях, например GitHub, CodeCommit и BitBucket. Поместив код в репозиторий, его можно тестировать, разветвлять, разрабатывать и объединять. Это позволяет более крупной команде постоянно работать и разрабатывать стек изолированно, не создавая конфликтов с предыдущими изменениями.

Существуют облачные сервисы, которые охватывают все аспекты процесса разработки кода так, как показано на рис. 8.3.

Рис. 8.3

Для платформы AWS сервисы, показанные на рис. 8.4, связаны с каждым из названных ранее процессов.

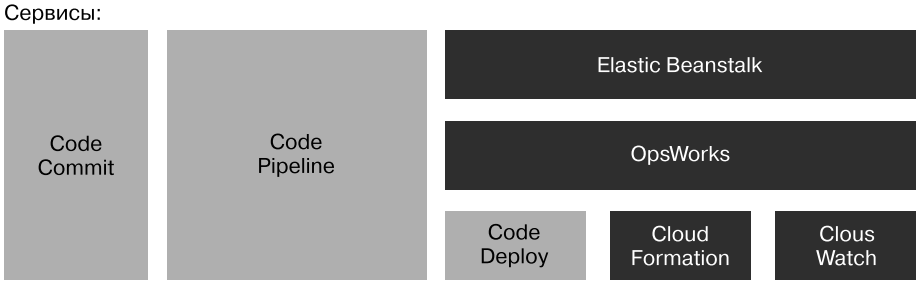


Рис. 8.4

Более подробный пример сервисов AWS рассмотрен в главе 9.

IaC-код можно проверить с помощью SDK, имеющихся у облачных поставщиков для распространенных IDE, таких как Eclipse. Более того, такие инструменты, как cfn-nag от Stelligent, можно использовать для автоматического обнаружения проблем в вашем коде перед развертыванием. Проект находится по адресу https://github.com/stelligent/cfn_nag. Cfn-nag можно использовать на этапе фиксации, это означает, что разработчик IaC перед развертыванием может быстро получить обратную связь. Можно вставить этот код в конвейер и запросить код выхода при обнаружении критических ошибок.

Такие сервисы, как Amazon Code Pipeline, предоставляют пользователям простой управляемый сервис для непрерывной интеграции и непрерывного развертывания. Code Pipeline создает, тестирует и развертывает код каждый раз, когда в вашем репозитории кода применяются новые изменения или активизируется веб-хук (механизм оповещения о событиях). Развертывание осуществляется с помощью сервиса AWS CodeDeploy, который автоматизирует развертывание программного обеспечения в вычислительных средах, доступных на заданной платформе, таких как EC2, AWS Lambda, и даже локальных виртуальных машин (рис. 8.5).

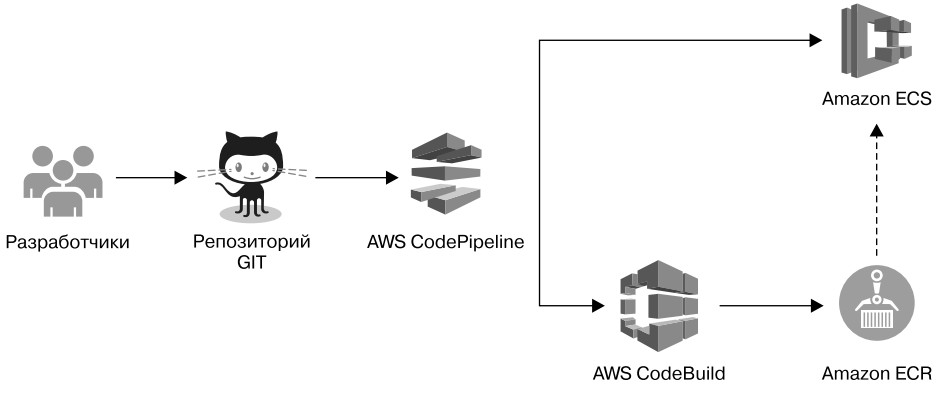


Рис. 8.5

Продвинутый облачный подход к разработке кода может выглядеть, как на данной схеме. Разработчики создают репозиторий кода на Git. После проверки кода AWS CodePipeline тестирует и отправляет исходный код в сервис AWS CodeBuild, который компилирует исходный код, запускает тесты, предоставляет пакеты программного обеспечения и развертывает их в Amazon *Elastic Container Registry (ECR)*. В дальнейшем ECR поддерживает копии вашего контейнерного приложения, которые развертываются в Amazon *Elastic Container Service (ECS)*.

После того как ваш конвейер завершит процесс от фиксации/отправки до развертывания, мониторинг и отслеживание конфигурации становятся первостепенными задачами эксплуатации облачных систем. Создание настраиваемых журналов и показателей, относящихся к конкретному приложению, сервису или микросервису, — это ключ к поддержанию стабильности в долгосрочной перспективе.

- Для приложения примером может быть время жизни пользовательского запроса (как долго пользовательские задания оставались в очереди).
- Для сервиса примером могут быть запросы сервиса на основе API для определения количества друзей у пользователя (например, приложение социальной сети).
- Для микросервиса примером может быть количество запросов контейнера, поступающих из всей корпоративной среды в микросервис оркестрации контейнеров.

Облачный инструментарий

Инструменты эксплуатации облачных систем могут охватывать множество различных областей, и следующий список далеко не исчерпывающий. В него включены некоторые из наиболее важных инструментов, которые мы использовали или встречали в своих проектах.

Slack

Большинство разработчиков считают использование электронной почты или любого другого сервиса обмена сообщениями плохой идеей. Slack представляет новый подход к взаимодействию внутри команды. Это унифицированная коммуникационная платформа для принятия решений, обмена наработками и управления системами, позволяющая разработчикам облачного ПО взаимодействовать между собой и с ботами (slack.com/).

Stelligent cfn-nag

Инструмент cfn-nag (github.com/stelligent/cfn_nag) ищет среди шаблонов AWS CloudFormation те, которые могут указывать на небезопасную инфраструктуру. Грубо говоря, он будет искать:

- слишком мягкие правила IAM (использование символов подстановки);
- слишком мягкие правила группы безопасности;

- не включенное журналирование доступа;
- не включенное шифрование.

GitHub

GitHub — это платформа разработки, которая позволяет размещать и просматривать код, управлять проектами и создавать программное обеспечение. Это, по большому счету, самая популярная платформа разработки для размещения кода, насчитывающая более 28 млн разработчиков (github.com/).

Резюме

В этой главе мы обсудили недостатки тех методов, с помощью которых организации традиционно выделяют рабочую силу для создания тех или иных систем. Эти недостатки становятся все более очевидными по мере того, как облачные технологии превращаются в серьезный бизнес-инструмент. Затем мы представили новую модель разработки, развертывания и обслуживания систем — CND-команды.

CND-команды находятся в центре разработки облачных архитектур с богатыми возможностями, поскольку они достаточно гибки, свободны и целеустремленны, чтобы нести полную ответственность за свои собственные системы. Их действия не скованы решениями коммитетов по утверждению изменений или надзором со стороны руководства; главное, чтобы они поддерживали свои API и SLA на том уровне, которого от них ожидают другие системы и команды.

Мы обсудили работу с поставщиками сервисов, управляемых облаком, и то, как облачный MSP может стать ступенькой на пути к повышению зрелости облачной среды. Центральное место в эксплуатации облачных сервисов занимает IaC. Мы представили примеры того, как создавать, тестировать, развертывать и поддерживать системы с его помощью. Наконец, мы остановились на популярных сегодня на рынке облачных инструментах, позволяющих поддерживать работу в облаке.

В следующей главе обсудим уникальные особенности Amazon Web Services.

9 Amazon Web Services

Компания Amazon Web Services (AWS) стала пионером в мире облачных технологий. Она начала предлагать свои услуги в 2006 году, когда термин «облачные вычисления» еще даже не придумали! Первыми из сервисов, запущенных на AWS, были Amazon *Simple Queue Service (SQS)*, Amazon *Simple Storage Service (S3)* и Amazon *Elastic Compute Cloud (EC2)*, и они работают по сей день. Сегодня же AWS, возможно, один из наиболее продвинутых и развитых поставщиков облачных вычислительных мощностей, предоставляющий свои услуги в 18 географических регионах, поддерживающий более 130 сервисов и анонсировавший грандиозные планы на запуск еще много чего в будущем. С течением времени компания AWS диверсифицировала свои услуги, создав ряд новых технологий и решений так, что между понятиями IaaS, PaaS и SaaS начали стираться границы. В этой главе мы обсудим некоторые из особенностей AWS и новые архитектурные шаблоны, которые помогут вам грамотно использовать облако, применяя множество различных сервисов AWS. Далее перечислены темы, которые будут освещаться в этой главе.

- Облачные сервисы AWS и понятия, связанные с CI/CD, бессерверная архитектура, контейнеры и концепция микросервисов, а также следующие сервисы:
  - AWS CodeCommit;
  - AWS CodePipeline;
  - AWS CodeBuild;
  - AWS CodeDeploy;
  - Amazon EC2 Container Service;
  - AWS Lambda;
  - Amazon API Gateway.
- Управление возможностями облачной архитектуры приложения на AWS и их мониторинг.
- Методы перехода от монолитной архитектуры приложения к облачной архитектуре на AWS.
- Примеры архитектур и код реализации CI/CD, бессерверной и микросервисной архитектур приложения.

Чтобы всегда получать последние объявления от AWS и новости о запуске новых сервисов, подпишитесь на следующие ресурсы:

- новые возможности AWS (aws.amazon.com/ru/new/);
- AWS News Blog (aws.amazon.com/ru/blogs/aws/).

Облачные сервисы AWS (ось 1 CNMM)

Для начала взглянем на основные сервисы, предоставляемые AWS, которые пригодятся для создания ваших облачных приложений.

Введение

У AWS богатый выбор сервисов, включая основные инфраструктурные компоненты: Amazon EC2 (виртуальные серверы в облаке), Amazon EBS (блочное хранилище для EC2), Amazon S3 (облачное хранилище объектов) и Amazon VPC (изолированные облачные ресурсы, использующие виртуализированные сети). Эти сервисы существуют уже многие годы и к тому же заточены под крупномасштабное развертывание ПО. Кроме того, они могут предложить широкий набор инструментов, позволяющих разнообразить варианты решений, поставляемых конечному заказчику, из которых можно составить наиболее подходящую комбинацию, наилучшим образом соответствующую специфике бизнес-требований. Например, Amazon EC2 предлагает более 50 различных типов инстансов, чтобы подстраиваться под различные степени нагрузки и области применения. И если заказчик желает развернуть *проект с высокопроизводительными вычислениями (High Performance Computing)*, то можно воспользоваться инстансами, *оптимизированными под вычисления*. В то же время, если у вас база данных NoSQL, которая очень нуждается в производительных операциях ввода-вывода с низкой задержкой доступа к хранилищу, то пригодятся инстансы, *оптимизированные под хранение информации*.

Стоит заметить, что AWS запускала новые типы инстансов одновременно с ежегодным обновлением процессоров и увеличением размера памяти, задействуемой приложениями, что позволяет клиентам пользоваться новейшими и наилучшими вычислительными конфигурациями, не беспокоясь о типичных для центров обработки данных циклов закупки и обновления оборудования. В то же время для обслуживания хранения и поддержки сетей Amazon EBS, Amazon S3 и Amazon VPC позволяют выстраивать множество различных конфигураций, обеспечивая гибкость там, где это необходимо.

Одними из важнейших преимуществ услуг поставщиков облака выступают эластичность (elasticity) и гибкость (agility), которые позволяют наращивать и со-

кращать инфраструктуру в соответствии с потребностями вашего приложения. Это кардинальное отличие от традиционного подхода с использованием центра обработки данных, когда нужно самостоятельно обслуживать пиковые нагрузки, из-за которых ресурсы инфраструктуры зачастую, с одной стороны, простаивали, а с другой — к непредсказуемым нагрузкам готовы не были. Эта ситуация резко изменилась с появлением облака, когда поставщики, такие как AWS, начали предоставлять инновационные сервисы наподобие AutoScaling, который динамически изменяет ваши вычислительные мощности в соответствии с поведением приложения, например, при изменении объема использования CPU или на основании каких-либо оригинальных показателей наблюдения за приложением. Кроме триггеров масштабирования, AWS может предложить автомасштабирование на основе шаблонов использования по времени, например, в соответствии с вариативностью использования по часам, дням, неделям. Наряду с вычислительным автомасштабированием AWS предлагает хранилище Elastic EBS, которое способно увеличиваться, повышать производительность или изменять тип хранилища на лету, прямо во время его использования. Такие типы сервисов определенно являются отличительной чертой облака и, по сути, сегодня стали нормой при создании веб-приложений. Тем не менее, чтобы воспользоваться некоторыми из этих возможностей, ваше приложение должно быть к ним готово с архитектурной точки зрения. Например, если вы хотите воспользоваться автомасштабированием для своего веб-сервера, чтобы компенсировать увеличение нагрузки на сайт, то одним из первых необходимых изменений с точки зрения приложения будет изменение ситуации с локальным хранением на каком-либо веб-сервере информации о состоянии сессии. Ее же нужно вывести в кэш (например, ElastiCache) или даже в базу данных (например, Amazon DynamoDB), чтобы можно было беспрепятственно масштабировать серверные мощности, не беспокоясь о прерывании сессий пользователей.

Пример архитектуры, опубликованной AWS, для создания автомасштабируемых веб-приложений с применением ряда основных сервисов представлен на рис. 9.1.

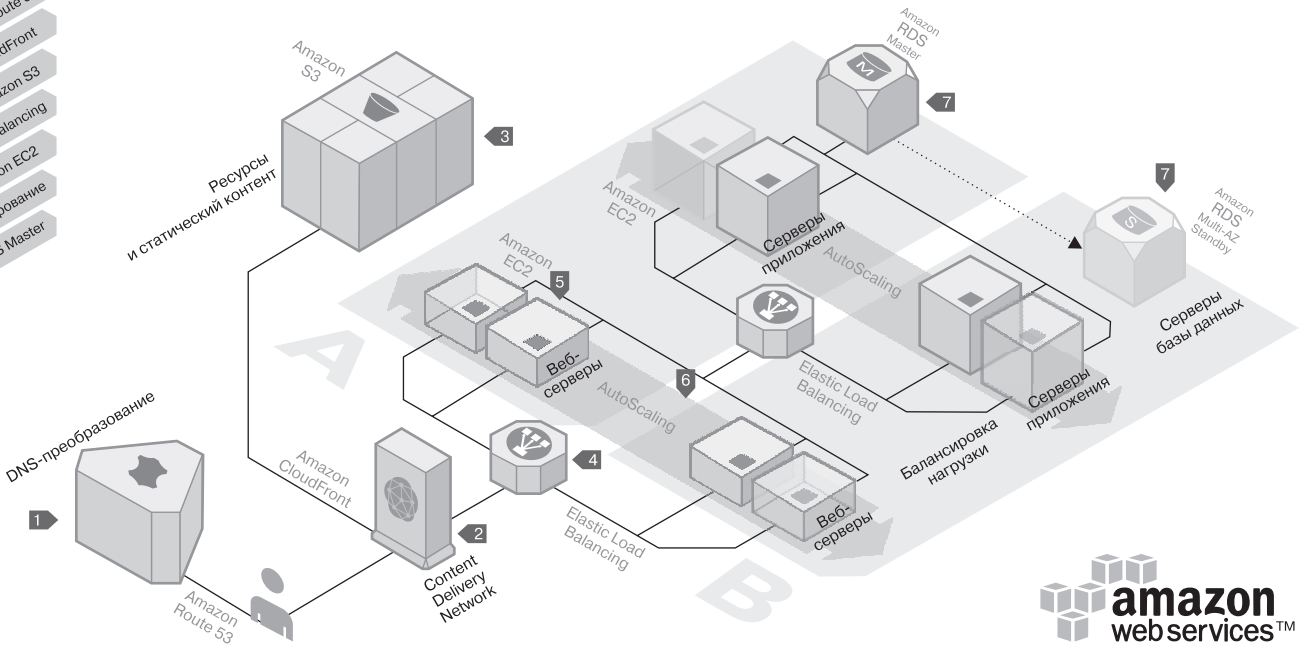
Теперь, после того как мы рассмотрели основные составляющие, стоит упомянуть, что Amazon предлагает и другие достойные сервисы, облегчающие задачу развертывания приложений для конечного пользователя, чтобы ему не приходилось беспокоиться о налаживании инфраструктуры. Таков, например, Amazon RDS — управляемый сервис для работы с базами данных MySQL, PostgreSQL, Oracle, SQL Server и MariaDB. Amazon RDS позволяет гибко настраивать реляционные базы данных и управлять ими, автоматизируя такие времязатратные задачи, как предоставление оборудования, установка базы данных, ее обновление и резервное копирование. Так что с его помощью вы довольно быстро можете начать развертывать свою базу данных, даже не обладая навыками администратора БД. Кроме того, AWS предлагает множество других сервисов, например AWS Elastic Beanstalk (для запуска и управления приложениями) и Amazon OpsWorks (для автоматизации администрирования с помощью Chef), которые называют высокоуровневыми сервисами из категории PaaS.

Руководство по AWS-архитектуре

- Amazon Route 53
- Amazon CloudFront
- Amazon S3
- Elastic Load Balancing
- Amazon EC2
- Автомасштабирование
- Amazon RDS Master

ХОСТИНГ ВЕБ-ПРИЛОЖЕНИЯ

Поддержка широко доступного и расширяемого веб-приложения может стать сложной и дорогой задачей. При необходимости обслуживать периоды пиковой нагрузки и колебания паттернов трафика мы попадаем в ситуацию, когда дорогое оборудование используется неэффективно. Amazon Web Services поставляют надежную, расширяемую, безопасную и высокопроизводительную инфраструктуру, необходимую веб-приложениям для обеспечения растяжимой, масштабируемой инфраструктуры, способную регулировать бюджет, выделяемый клиентом, в реальном времени с учетом колебаний трафика



amazon web services™

Обзор системы

1. Пользовательские DNS-запросы обслуживаются **Amazon Route 53**, сервисом DNS с высокой степенью доступности. Сетевой трафик перенаправляется в инфраструктуру, развернутую на AmazonWeb-Services.

2. Статический, потоковый и динамический контент поставляется **Amazon CloudFront**, глобальной сетью граничных узлов. Запросы автоматически перенаправляются к ближайшему граничному узлу, так чтобы поставка контента осуществлялась с наилучшей производительностью.

3. Ресурсы и статический контент, используемый веб-приложением, хранится на **Amazon Simple Storage Service (S3)**, высокоустойчивой инфраструктуре хранилища, предназначенного для критически важных задач хранения данных.

4. HTTP-запросы сначала передаются **Elastic Load Balancing**, который автоматически распределяет входящий в приложение трафик между множеством экземпляров **Amazon Elastic-ComputeCloud (EC2)**, находящихся в пределах зон доступа (Availability Zones). Это же, в свою очередь, позволяет еще больше сократить вероятность отказа вашего приложения и беспрепятственно предоставлять необходимую долю распределения нагрузки в ответ на изменение поведения входящего трафика приложения.

5. Веб-серверы и серверы приложений развертываются на экземплярах Amazon EC2. Большинство организаций предпочитают **Amazon Machine Image (AMI)** и затем подстраивают их под свои нужды. Настроенный сервис AMI позже станет точкой отсчета для будущих преобразований.

6. Веб-серверы и серверы приложений развертываются в группу **автомасштабирования**. Автомасштабирование автоматически повышает или понижает емкость вашей системы в соответствии с условиями, которые вы определили. Благодаря этому вы можете быть уверены в том, что количество ваших инстансов **Amazon EC2** всегда увеличивается во время всплесков нагрузки, поддерживая определенный уровень производительности, и автоматически уменьшается, когда в них нет острой нужды, чтобы минимизировать расходы.

7. С целью обеспечения высокой доступности реляционная база данных, в которой находятся ресурсы приложения, развертывается в **Amazon Relational Database Service** (Amazon RDS) с резервированием в нескольких зонах доступности.

Рис. 9.1

Используя их, вы можете быстро запустить свои приложения на AWS, не особо вдаваясь в детали процесса налаживания инфраструктуры. С помощью AWS OpsWorks вы сможете получить еще больше рычагов управления. Например, если хотите автоматизировать специфический процесс настройки или запуска приложения, можете использовать возможности AWS и реализовать это с помощью Chef.

AWS может предложить и другие PaaS-сервисы в сферах аналитики, больших данных и AI, например Amazon EMR (устанавливает фреймворк Hadoop), Amazon Kinesis (работает с потоковыми данными в реальном времени), Amazon Lex (собирает голосовых и текстовых ботов), Amazon Rekognition (занимается поиском и анализом изображений) и Amazon Polly (преобразует текст в человеческую речь). Пользуясь такими типами сервисов, разработчики и архитекторы данных могут с легкостью создавать их приложения, больше сосредотачиваясь на бизнес-логике, чем на аспектах управления необходимой инфраструктурой.

С течением времени AWS резко продвинулась в сфере предложения совершенно новых типов сервисов, близких к SaaS. В основном это сервисы из категории обеспечения бизнес-продуктивности, такие как Amazon Chime (сервис для видеозвонков и чата), Amazon WorkDocs (корпоративное хранение и распространение информации) и Amazon Connect (сервис для контакт-центра). Это тоже довольно новая область для AWS, но с укреплением инфраструктуры эти сервисы станут пользоваться спросом и привлекательны будут не только с точки зрения инфраструктуры, но еще и как программный продукт с моделью оплаты по мере использования.

Платформа AWS: ключевые инструменты

Как говорилось в главе 1, CNMM — это одна из моделей зрелости, придерживаясь которой можно полагать, что если вы используете базовые возможности облака, такие как компоненты инфраструктуры, то ваш проект уже можно назвать облачным. Но этого может быть недостаточно, так как есть возможность использовать инструменты посерьезнее, чтобы полностью раскрыть потенциал облака. Руководствуясь этим, можно полагать, что если вы намерены по максимуму воспользоваться возможностями облака AWS, то вам стоит обратить внимание на некоторые инструменты.

Сервисы KRADL

Как говорилось ранее, AWS может предложить множество инновационных сервисов, которые помогут спроектировать архитектуру для множества областей применения и бизнес-кейсов. Существует ряд сервисов, определяемых в AWS как ключевые, и AWS настоятельно рекомендует вам использовать именно их, а не пытаться самим создавать аналогичные решения, так как они могут оказаться

нерасширяемыми, не с такой богатой функциональностью и не такими мощными, как решения AWS.

Все эти решения вместе называют *KRADL*, что расшифровывается следующим образом:

- *K* — Amazon Kinesis;
- *R* — Amazon Redshift;
- *A* — Amazon Aurora;
- *D* — Amazon DynamoDB;
- *L* — AWS Lambda.

Давайте более подробно рассмотрим каждый из этих пунктов, опираясь на то, как эти сервисы описывает сама AWS.

Amazon Kinesis облегчает задачу сбора, обработки и анализа в реальном времени потоковых данных, чтобы вы могли своевременно получать нужную информацию и быстро реагировать в соответствии с ней. С помощью Amazon Kinesis вы можете обрабатывать в реальном времени, например, журналы приложения, историю посещений веб-сайта, телеметрические данные и еще много чего в своих базах данных, озерах данных (data lakes) и хранилищах данных, а также создавать приложения, которые будут работать с этими данными в реальном времени. Amazon Kinesis позволяет вам обрабатывать и анализировать данные по мере их получения и действовать в реальном времени, вместо того чтобы ждать окончания сбора информации перед обработкой. Подробности — по адресу aws.amazon.com/ru/kinesis/.

Amazon Redshift — это скоростное полностью управляемое хранилище данных, которое упрощает и удешевляет анализ всех ваших данных с помощью стандартного SQL и существующих инструментов *BI* (Business Intelligence). Оно позволяет применять сложные аналитические запросы на петабайтах структурированных данных, использовать продвинутую оптимизацию запросов, хранить данные в столбцовых БД на высокопроизводительных локальных дисках и осуществлять высокопараллельное выполнение запросов. Подробности — по адресу aws.amazon.com/ru/redshift/.

Amazon Aurora — это ядро для работы с реляционными базами данных, совместимое с MySQL и PostgreSQL. Его возможности сочетают в себе скорость работы и высокие показатели доступности престижных коммерческих баз данных с простотой и нулевой стоимостью бесплатных решений. Производительность Aurora в пять раз выше, чем MySQL, вдобавок вы получаете уровень безопасности, доступности и надежности коммерческой базы данных за десятую часть стоимости оной. Amazon Aurora — это управляемый сервис для баз данных, созданный на основе распределенной и самовосстанавливающейся системы хранения, которой вы можете доверить свои данные, не опасаясь за их безопасность. Подробности — по адресу aws.amazon.com/ru/rds/aurora/.

Amazon DynamoDB — это быстрый и гибкий сервис для работы с базами данных NoSQL, применимый ко всем приложениям, которые нуждаются в постоянной при любом масштабировании стабильной задержке не более единиц миллисекунд. Это полностью управляемая облачная база данных, которая поддерживает как документную модель хранения, так и модель «ключ — значение». Это гибкая модель хранения данных с надежной производительностью и возможностью выполнять автоматическое масштабирование пропускной способности, что делает ее прекрасным кандидатом для использования в мобильных, веб-, игровых, технических, IoT и многих других приложениях. Подробности — по адресу aws.amazon.com/ru/dynamodb/.

AWS Lambda позволяет запускать код, не занимаясь выделением или администрированием серверов. Вы платите лишь за потребляемое вычислительное время — не нужно платить за период, когда код не работает. С помощью AWS Lambda вы можете писать код буквально для любого приложения или сервиса на бэкенде — и все это не требует администрирования. Просто загрузите свой код, и Lambda позаботится обо всем, что нужно для его запуска и масштабирования, позволяя ему быть легкодоступным. Вы можете настроить код на автоматический запуск из других сервисов AWS или на их вызов напрямую из мобильных или веб-приложений. Подробности — по адресу aws.amazon.com/ru/lambda/.

Большинство перечисленных сервисов AWS — проприетарные, поэтому иногда пользователи задумываются о том, что они останутся зависимыми от AWS. Тем не менее реальность такова, что даже в локальной среде заказчики используют множество пакетных решений (например, базы данных и ERP), от которых потом трудно отказаться. Если ту же логику применить в отношении облачных сервисов, зависимость от поставщика является скорее вопросом восприятия, нежели чем-то реальным. На самом деле сервисы Amazon Aurora и Amazon Redshift предлагают те же интерфейсы, что и PostgreSQL/SQL, и при желании вы сможете с легкостью перенести свои приложения на другую платформу. В то же время если заказчик захочет создать подобные сервисы самостоятельно, то это, вероятно, отнимет время и ресурсы, и не только на создание, но еще и на администрирование и итеративное добавление новой функциональности, а это как раз то, чем занимается AWS. Поэтому не лучше ли будет сосредоточиться на основной бизнес-логике и переложить все остальное на плечи сервисов AWS.

Стоит взглянуть на KRADL-сервисы платформы AWS и с точки зрения их уникальности по сравнению с сервисами, предлагаемыми другими поставщиками. Эти поставщики начали предлагать аналоги некоторых из них, но AWS были первопроходцами и уже давно развивают свои сервисы, поэтому их решения более продвинутые.

Взгляните на интересную инфографику от 2ndWatch — консалтингового партнера AWS, отображающую анализ облачных сервисов AWS: https://www.2ndwatch.com/wp-content/uploads/2017/05/Cloud-Native-Services.pdf.

Сервисы безопасности AWS

Пользователям облачных сервисов важно поддерживать правильную стратегию безопасности, соответствующую рабочей нагрузке и корпоративной политике управления. Временами бывает сложно перенести инфраструктуру из локальной среды в облако, это даже не совсем сопоставимые понятия. Тем не менее не так давно AWS запустила множество новых сервисов специально для обеспечения безопасности, чтобы сократить пропасть между возможностями локальной среды и AWS.

Уже несколько лет у Amazon работает сервис *IAM* (Identity and Access Management), с помощью которого вы можете управлять своими облачными пользователями и группами, а также их правами доступа к различным ресурсам AWS. С помощью этого сервиса можно настроить продуманную систему мер безопасности (например, наложить ограничения на специфичные API-запросы в вашей среде и давать разрешения только определенным IP-адресам), интегрироваться с корпоративным каталогом, чтобы обеспечить общий доступ, а также задействовать лишь *многофакторную аутентификацию (MFA)*, требующуюся при определенных действиях в отношении каких-либо сервисов.

Сегодня IAM — жизненно необходимый сервис для управления средой AWS. Кроме этого, часто забывают о двух других невероятно полезных облачных сервисах для любого типа окружений.

- *AWS Key Management Service*. Это управляемый сервис, который облегчает задачу создания и обслуживания ключей, используемых для шифрования ваших данных, он также задействует *Hardware Security Modules (HSM)* для защиты ваших ключей. AWS Key Management Service интегрирован с несколькими другими сервисами AWS, чтобы вам было легче защищать данные, хранящиеся в ваших сервисах. Подробнее — по адресу aws.amazon.com/ru/kms/.
- *AWS CloudTrail*. Это сервис, который позволяет вам администрировать свой AWS-аккаунт, контролировать его, а также выполнять аудит эксплуатации и рисков. С помощью CloudTrail вы можете собирать журналы, вести непрерывный мониторинг и сохранять сведения о событиях, связанных с API-запросами по всей вашей инфраструктуре AWS. CloudTrail предоставляет историю API-запросов для вашего аккаунта, включая те, которые отправлялись через AWS Management Console, AWS SDK, консольные инструменты или другие сервисы AWS. Подробнее — по адресу aws.amazon.com/ru/cloudtrail/.

Эти сервисы способны предложить полную управляемость, интеграцию с различными сервисами AWS и легкость в применении. По сути, с такими легкими в использовании и настраиваемыми сервисами любой стартап, пользуясь платформой AWS, получает в свое распоряжение те же средства администрирования, что имеют корпоративные клиенты. Это настоящая сила демократизации, которую делает возможной облако, когда все сервисы и инструменты становятся общедоступными, тем самым открывая возможности для инноваций и создания новых моделей для приложений.

С приходом на рынок таких облачных решений клиентам больше нет необходимости самим поддерживать массивные инструменты для обслуживания ключей, покупать или создавать оригинальные программные пакеты, которые выполняли бы эти функции, крайне необходимые для процессов развертывания. Однако некоторые из этих сервисов доступны только в облачной среде, поэтому, если вы управляете гибридной средой, включающая в себя локальные инфраструктурные компоненты, то использование единой консоли для администрирования и мониторнига этой среды может стать испытанием.

Довольствуясь основными возможностями, предоставляемыми сервисами в плане функциональности, вы можете, используя современные архитектурные шаблоны, создавать самообучаемые и самоадаптирующиеся модели обеспечения безопасности. Например, если вы подключите к своему аккаунту журналирование с CloudTrail, то сможете обнаруживать какую-либо нежелательную активность или недобросовестное использование ресурсов вашего аккаунта AWS и прописывать динамично адаптирующиеся методы реагирования. Для обеспечения такой возможности прекрасно подходит AWS Lambda, с помощью которой вы можете настроить логику не просто на обнаружение, а на характерное реагирование, основанное на определенных условиях. Эту возможность позднее можно будет сочетать с продвинутыми техниками, такими как машинное или глубокое обучение, и вместо того, чтобы прописывать особые условия для реагирования, вы можете создать самообучающуюся модель, которая будет формировать такие типы адаптирующихся систем безопасности. Все это становится возможным с помощью сервисов и инструментов, предоставляемых облаком.

Кроме AWS KMS и AWS CloudTrail, AWS может предложить множество новых сервисов для обеспечения безопасности, которые помогут вам в определенных ситуациях.

- *Amazon Inspector*. Это автоматизированный инструмент оценки безопасности, который улучшает безопасность приложений, развернутых на AWS, и приводит их в соответствие стандартам. Amazon Inspector автоматически оценивает уязвимости приложения и расхождение их методов реализации с принятыми практиками. После этого он составляет подробный список угроз, выстроенных по уровню серьезности. Результаты можно просмотреть напрямую в Amazon Inspector, или в составе отчета, доступного в консоли Amazon Inspector, или через API. Подробнее — по адресу aws.amazon.com/ru/inspector/.
- *AWS Certificate Manager*. Это сервис, который облегчит вам задачу по обслуживанию, управлению и развертыванию в работе с сертификатами *Secure Socket Layer (SSL)/Transport Layer Security (TLS)* для сервисов AWS. Сертификаты SSL/TLS применяются для защиты сетевых коммуникаций и идентификации веб-сервисов в Интернете. AWS Certificate Manager избавляет вас от необходимости тратить время и усилия на приобретение, загрузку и обновление сертификатов SSL/TLS.

- *AWS WAF*. Это брандмауэр веб-приложений, который помогает защитить ваши приложения от рядовых эксплойтов, способных повлиять на доступность приложений, скомпрометировать безопасность или злоупотреблять вашими ресурсами. AWS WAF позволяет настраивать специфические правила безопасности, чтобы управлять трафиком и разрешать или запрещать действия приложений. Вы можете использовать AWS WAF для того, чтобы создавать собственные правила для защиты от распространенных видов атак, таких как SQL-инъекции или межсайтовые скрипты. Подробнее — по адресу aws.amazon.com/ru/waf/.
- *AWS Shield*. Это управляемый сервис для защиты приложений, работающих на AWS, от *DDoS*-атак. AWS Shield позволяет непрерывно отслеживать угрозы и автоматически избегать их, чтобы минимизировать время простоя и задержек и, таким образом, избавиться от необходимости обращаться в AWS Support. Подробнее — по адресу aws.amazon.com/ru/shield/.
- *Amazon GuardDuty*. Это управляемый сервис для обнаружения угроз, который позволяет вам легче и точнее осуществлять постоянный мониторинг и защиту своих аккаунтов AWS и приложений. Подробнее — по адресу aws.amazon.com/ru/guardduty/.
- *Amazon Macie*. Это сервис для обеспечения безопасности, использующий машинное обучение для обнаружения, классификации угроз и защиты важных данных. Подробнее — по адресу aws.amazon.com/ru/macie/.

Самое большое преимущество перечисленных облачных сервисов состоит во том, что вы можете начать использовать их в любой момент, не беспокоясь о приобретении лицензии, сложном конфигурировании и т. д. Но поскольку эти сервисы все еще остаются новинкой по сравнению с некоторыми корпоративными программными ISV-пакетами с аналогичной функциональностью, то для более сложных случаев могут не вполне подойти. Для таких случаев в AWS есть AWS Marketplace, где размещается множество ISV-партнеров, предлагающих решения программных пакетов, совместимых с облаком, которые можно легко и быстро развернуть в окружении AWS. Так что в зависимости от области применения и того, какие требуются инструменты, рекомендуется сначала оценить облачные сервисы AWS, а потом при необходимости взглянуть на ISV-решения.

Машинное обучение/искусственный интеллект

В последние несколько лет AWS демонстрировала свой серьезный вклад в возможности работы с машинным обучением (ML) и искусственным интеллектом (AI). Во время одной из ежегодных конференций (AWS re:Invent) в 2017 году руководство AWS сделало несколько громких заявлений о своих решениях в сфере ML/AI, которые с того времени приобрели большую популярность. Далее приводятся некоторые из ключевых сервисов из портфолио AWS в рамках возможностей ML/AI.

- *Amazon SageMaker*. Он позволяет разработчикам и специалистам по обработке данных быстро и легко создавать, обучать и развертывать модели машинного обучения с помощью высокопроизводительных алгоритмов и поддержки широкого спектра платформ, пользуясь быстрыми процессами обучения, настройки и получения результатов. Amazon SageMaker создавался по модульной архитектуре, так что можете задействовать часть или все его возможности в процессах с машинным обучением. Подробности — по адресу aws.amazon.com/ru/sagemaker/.
- *Amazon Recognition*. Это сервис, который помогает дополнять ваши приложения мощными инструментами для визуального анализа. Recognition Image позволяет с легкостью создавать приложения для поиска, проверки и организации миллионов изображений. Еще он позволяет извлекать динамичный контекст из статических видеофайлов или видеотрансляций и помогает анализировать их. Подробнее — по адресу aws.amazon.com/ru/rekognition/.
- *Amazon Lex*. Это сервис для создания интерфейсов для общения с помощью голоса и текста. Основанный на том же движке, что и Alexa, он обеспечивает высококачественное распознавание речи и языка, что позволяет использовать продуманных чат-ботов. Подробнее — по адресу aws.amazon.com/ru/lex/.
- *Amazon Polly*. Это сервис, преобразующий текст в естественную речь. Он позволяет приложениям говорить и дает возможность создавать совершенно новые продукты, использующие речь, которые будут применяться везде, от мобильных телефонов и машин до девайсов и инструментов. Подробности — по адресу aws.amazon.com/ru/polly/.

Кроме представленных сервисов, AWS поддерживает множество популярных фреймворков и библиотек машинного обучения, которые регулярно используются специалистами по данным и разработчиками. Среди них Apache MXNet, TensorFlow, PyTorch, Microsoft *Cognitive Toolkit (CNTK)*, Caffe, Caffe2, Theano, Torch, Gluon и Keras. Все это предлагается в рамках *Amazon Machine Image (AMI)*, поэтому начать работать с ними можно, сделав всего несколько щелчков кнопкой мыши. Вместе с вычислительными GPU-инстансами и инстансами программируемой логической матрицы типа *FPGA* можно ускорить обработку сложных алгоритмов и моделей, тем самым убеждаясь в том, что платформа способна проявить себя в любых областях применения ML/AI.

Хранение объектов (S3, Glacier, экосистема)

AWS может предложить разного рода сервисы для хранения, например блочное хранилище, файловое хранилище, хранилище объектов и архивное хранилище. Несмотря на то что все эти сервисы работают независимо друг от друга и этим преимуществом можно воспользоваться, клиенты часто комбинируют эти сервисы, чтобы создавать различные варианты для организации хранилищ. Например, Amazon EBS — это блочное хранилище, связанное с инстансами Amazon EC2 для

локального хранения и обработки данных. Когда бы ни понадобилось распределить данные между множеством инстансов EC2 в виде файлового каталога, Amazon EFS будет кстати, и этот же EFS можно применять с другими инстансами EC2 с помощью интерфейса NFS v4.1.

Сегодня, когда данные требуется подвергать резервному копированию, что предполагает долгосрочное и надежное хранение, Amazon S3 будет как нельзя более кстати, ведь он может хранить не просто данные, а целые снимки томов EBS, которые будут храниться в разных зонах доступности (Availability Zones) с поддержкой резервирования. Так что, используя этот механизм, можно разработать полный жизненный цикл управления данными, задействуя множество сервисов для хранения.

Помимо Amazon EBS образы *Amazon Machine Images (AMI)*, с помощью которых запускаются инстансы Amazon EC2, по умолчанию хранятся также на Amazon S3. И журналы, например, от Amazon CloudTrail или Amazon CloudWatch, хранятся в Amazon S3. Даже у сервисов для миграции от AWS, таких как AWS Snowball и AWS Database Migration Service, настроена интеграция с Amazon S3. Для других сервисов, предназначенных для работы с большими данными, таких как Amazon EMR, Amazon Redshift или даже Amazon Antenna, Amazon S3 тоже является ключевым компонентом системы. Он выступает в качестве источника и/или пути назначения для потока данных, сортируемых и преобразуемых для решения разного рода аналитических проблем. Подобным образом многие другие сервисы AWS скрыто пользуются Amazon S3 для хранения объектов или резервного копирования.

В итоге получается, что Amazon S3 — один из ключевых компонентов AWS и он должен стать частью любого облачного продукта на AWS. Кроме озвученного ранее ряда интеграций, Amazon S3 готов предложить также множество функций, вносящих гибкость в создание решения для приложения. Например, в Amazon S3 существует множество классов хранения, и в зависимости от вашей модели доступа к объектам и требований к надежности и доступности вы можете использовать классы из следующей таблицы.

Класс хранения	**Долговечность (durability), %**	**Доступность (availability), %**	**Применение**
STANDARD	99,999999999	99,99	Идеально подходит для приложений, в которых важна производительность и доступ к данным осуществляется часто
STANDARD_IA	99,999999999	99,99	Оптимизирован для работы с длительно хранящимися и нечасто используемыми данными, например резервными копиями, или другими данными, к которым нет необходимости часто давать доступ, но все еще требуется высокая производительность

Класс хранения	Долговечность (durability), %	Доступность (availability), %	Применение
Reduced Redundancy Storage (RRS)	99,99	99,99	Предназначается для неключевых воспроизводимых данных, которые хранятся на более низких резервных уровнях, чем у класса STANDARD

Amazon S3 имеет ряд функций, связанных с безопасностью, таких как шифрование (на стороне сервера или клиента), версионирование объектов для сохранения истории, защита от удаления объектов с MFA.

У Amazon S3 много ISV-партнеров, которые либо непосредственно интегрируются с ним, либо используют его в качестве источника или места назначения при реализации решений для хранилищ. Вам стоит взглянуть на решения партнеров, пройдя по ссылке aws.amazon.com/ru/backup-recovery/partner-solutions/. Это поможет присмотреться к существующим ISV, для которых Amazon S3 (и другие сервисы для хранения) является составляющей решения.

Основываясь на сказанном ранее, можно отметить, что Amazon S3 — это один из основных сервисов для составления множества типов архитектур, и его определенно не стоит упускать из виду при проектировании любого рода процессов развертывания. Далее приводятся некоторые способы применения Amazon S3 в облачной архитектуре для поддержки множества пользователей.

- *Веб-приложения*. Amazon S3 может служить хранилищем для статического контента, такого как видео, изображения, HTML-страницы, CSS и разного рода скрипты для выполнения на стороне клиента. Поэтому если ваш сайт статичен, то веб-сервер и не понадобится, можете просто задействовать Amazon S3! Если у вас имеется глобальная база пользователей и вы хотите кешировать часто запрашиваемый контент поближе к их расположению, то прекрасной комбинацией для этого станет Amazon S3 с Amazon CloudFront.
- *Хранение журналов*. В рамках end-to-end-системы существует множество видов журналов, которые генерируются различными приложениями и компонентами инфраструктуры. Их можно хранить локально на дисках и анализировать с помощью инструментов Elasticsearch, Logstash и Kibana (ELK). Тем не менее множество клиентов используют Amazon S3 для сбора всех журналов, куда данные стекаются с помощью бесплатных решений, таких как Fluentd (www.fluentd.org), AWS Services (например, CloudWatch Logs или Amazon Kinesis), или коммерческих ISV-решений (например, Sumologic), которые можно поискать в том же Marketplace. Amazon S3 прекрасно подойдет для работы с любого рода журналами вне зависимости от того, будет ваше решение заниматься потоковой передачей или анализом данных. К тому же со временем, когда журналы устареют, их можно будет либо удалить, либо отправить на более дешевое хранение (Amazon S3-IA или Amazon Glacier), настроив политику их жизненного цикла.

- *Озера данных.* Amazon S3 — это сервис, который прекрасно подходит для создания масштабируемых озер данных, где можно хранить подготовленные, обработанные или даже промежуточные данные в качестве составляющей конвейера обработки данных. AWS дает пример архитектуры и модуля автоматизированного развертывания, которые демонстрируют пользу Amazon S3 как центрального сервиса в таком сценарии. Можете почитать об этом здесь: docs.aws.amazon.com/solutions/latest/data-lake-solution/welcome.html.
- *Событийно-ориентированные архитектуры.* API и слабосвязанные архитектуры сегодня встречаются повсеместно. Вместе с API популярность набирают событийно-ориентированные архитектуры (even-driven architecture, EDA), где с помощью простого вызова API вы можете инициировать на стороне сервера цепочку рабочих процессов. Сейчас существует множество компонентов для хранения, которые можно подключить к такого рода архитектурному шаблону, но Amazon S3 может предложить некоторые ключевые компоненты для нескольких областей применения.
  - Представьте, что у вас есть API или архитектура, управляемая событиями, которые нужны для запуска конвейера обработки медиаданных. Именно для хранения всех этих сырых объектов данных можно использовать Amazon S3. Таким же образом можно с помощью Amazon CloudFront передавать в S3 финальный вывод видео- и аудиоданных и изображений в ходе постобработки.
  - Координировать различные сервисы можно с помощью Amazon S3, выступающего в качестве хранилища, и запускающего механизма. Представим, что есть сервис А, который запускает обработку объектов в корзине А[1]. После этого объект записывается в корзине Б, а она посылает уведомление о событии, например, событию `s3:ObjectCreated:*`, которое затем подхватывают функции Amazon Lambda, чтобы запустить следующий шаг. Таким образом можно создавать слабосвязанные конвейеры, в которых координационный механизм основан на S3.
  - Возможна пакетная обработка данных — AWS запустила сервис для пакетных вычислительных процессов, который называется AWS Batch. И хотя у него широкий функционал, чаще всего его используют для анализа больших данных. В итоге Amazon S3 становится ключевым сервисом, который способен хранить и обрабатывать объекты любого объема. Например, на диаграмме с сайта AWS (рис. 9.2) демонстрируется применение пакетных вычислений на примере из индустрии финансовых сервисов, где данные из нескольких источников собирают в конце дня, чтобы составить отчеты и понять поведение рынка в течение дня. Таким образом, Amazon S3 стал в некотором роде основным инструментом хранения для различных уровней обработки.

[1] Корзина — это контейнер для объектов (см. https://aws.amazon.com/ru/s3/getting-started/). — *Примеч. пер.*

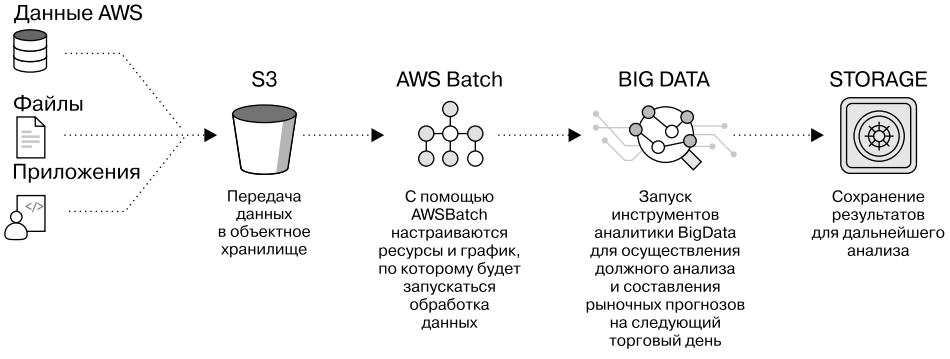


Рис. 9.2

Проектирование, ориентированное на приложения (ось 2 CNMM)

В предыдущем разделе мы познакомились с некоторыми ключевыми облачными сервисами, предоставляемыми AWS. Здесь же взглянем на другой аспект CNMM при создании облачных архитектур AWS. Существует множество типов архитектур и подходов, но мы сосредоточимся на двух ключевых шаблонах — бессерверной архитектуре и микросервисах. На самом деле эти шаблоны связаны между собой, так как сервисы, которые помогают создавать бессерверные решения, применимы и для разработки более мелких отлаженных сервисов, выполняющих единственную функцию. В следующем подразделе узнаем, как создавать микросервисы, которые по своей природе соответствуют шаблону бессерверной архитектуры.

Микросервисы бессерверной архитектуры

Ключевая идея микросервисов в бессерверной архитектуре состоит из трех шагов (рис. 9.3).

Давайте рассмотрим эту схему более детально.

API-триггер

Микросервис можно вызывать двумя способами:

- вручную, непосредственно взаимодействуя с веб-интерфейсом портала (или через командную строку);
- с помощью автоматизированного приложения или другого микросервиса, который вызывает этот сервис из цепочки управляемого рабочего процесса.

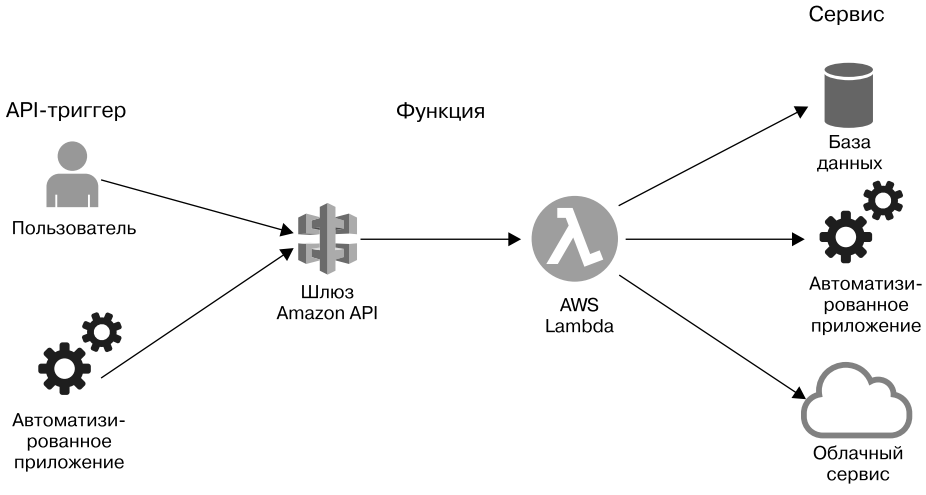


Рис. 9.3

API можно вызывать также с помощью системных событий, используя триггеры, запускающие соответствующие скрипты.

Функция

API-вызов, выполняемый на уровне триггеров, принимается шлюзом Amazon API Gateway, который поддерживает управление трафиком, авторизацию, контроль доступа, валидацию, трансформацию и мониторинг, за которыми следует вызов бэкенда, в данном случае размещаемого на AWS Lambda.

AWS Lambda — это бессерверная вычислительная среда, вы просто загружаете в нее свой код (Node.js, Python, Java или C#), задаете триггеры, а все остальное делает сервис. Таким образом, как и в предыдущем примере архитектуры, шлюз API будет вызывать функцию Lambda, чтобы исполнить там свою бизнес-логику. В этой бизнес-логике вы сможете реализовать любые действия, и об этом рассказывается в следующем подразделе.

Сервис

В рамках функции AWS Lambda можно реализовать любые действия, но чаще всего клиенты используют его в следующих случаях.

- Для реализации всей обработки в рамках функции AWS Lambda, что может подразумевать обработку данных или трансформацию, локальное выполнение скриптов и т. д. А в завершение — сохранение результатов в долговременное

хранилище. Для таких сценариев идеально подходит Amazon DynamoDB, который, в свою очередь, расширяет бессерверную архитектуру (не требуется обслуживать хост и ее легко интегрировать с AWS Lambda и API/SDK).

- Для взаимодействия со сторонними сервисами или даже с другими микросервисами, работающими в этой же среде, чтобы вызвать эти сервисы. Вызовы можно делать синхронными или асинхронными в зависимости от конкретного случая.
- Для интеграции с другими облачными сервисами с целью выполнения каких-то других действий. Например, представьте, что вы хотите запустить создание стека LAMP через API, в этом случае будете вызывать CloudFormation из своего кода в Lambda, чтобы инициализировать новую среду со стеком LAMP.

Здесь приведены лишь несколько примеров создания бессерверных микросервисов в AWS, но возможности этой технологии практически безграничны.

Пример бессерверного микросервиса

Теперь, когда вы изучили основы бессерверных микросервисов в AWS, давайте поупражняемся, работая с пробным приложением, использующим несколько сервисов AWS.

Создадим настоящий микросервис для получения информации о погоде из любой точки мира. Для этого будем применять архитектурные компоненты, представленные на рис. 9.4.

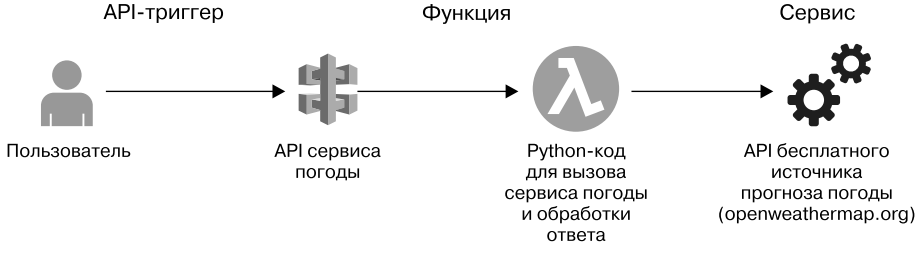


Рис. 9.4

Создание и настройка функции AWS Lambda

Войдите в консоль AWS и выберите Lambda в категории сервисов Compute. Затем следуйте инструкциям для создания функции Lambda.

1. Нажмите кнопку Create function.
2. Выберите Blueprint, а затем шаблон microservice-http-endpoint-python3.

3. Щелкните кнопкой мыши на Configure и укажите приведенные далее значения (они будут использоваться для настройки точки доступа шлюза Amazon API для этого микросервиса):
   - для API Name — Serverless Weather Service;
   - для Deployment Stage — prod;
   - для Security — Open. В таком случае мы сможем предоставлять доступ к API без какой-либо аутентификации (для другого рода проектов рекомендуется включать меры безопасности в работу с API либо с помощью AWS IAM, либо через опцию Open with).
4. В Create function укажите следующие значения.
   - Базовая информация:
     1) имя функции (например, `WeatherServiceFunction`);
     2) выберите Python 3.6 в списке Runtime;
     3) для Description укажите cloud-native — Serverless Weather Service.
   - Код функции Lambda: в поле Code Entry Type выберите Upload a .zip file. Возьмите образец Python-кода в ZIP-файле, `Cloudnative-weather-service.zip`, по ссылке https://github.com/PacktPublishing/Cloud-native-Architectures и загрузите его.
   - Обработчик функции Lambda и роль:
     1) для обработчика укажите `lambda_function.lambda_handler`;
     2) для роли выберите Create new role from template(s);
     3) для Role Name укажите WeatherServiceLambdaRole;
     4) для Policy Templates выберите значение Simple Microservice Permissions из выпадающего меню.
   - Теги: укажите для Key — `Name` и для Value — `Serverless Weather Microservice`.
   - Другие настройки:
     1) укажите для Memory — 128 MB;
     2) укажите для Timeout — 0 минут 30 секунд;
     3) оставьте остальные настройки неизменными.
5. После всего этого просмотрите настройки на основном экране и щелкните кнопкой мыши на Create function. В течение нескольких минут функция Lambda будет создана, и вы сможете увидеть ее в списке Functions (рис. 9.5).

В логике кода Lambda выполняются следующие три действия.

- Получаются параметры `zip` и `appid` из полученного запроса.
- Осуществляется `GET`-вызов к API OpenWeatherMap с параметрами `zip` и `appid`.

Рис. 9.5

- Получается ответ, который пересылается обратно в API Gateway, чтобы передать его конечному пользователю.

Сам код:

```
import boto3
import json
from urllib.request import Request, urlopen
from urllib.error import URLError, HTTPError

print('Loading function')

def respond(err, res=None):
    return {
        'statusCode': '400' if err else '200',
        'body': err if err else res,
        'headers': {
            'Content-Type': 'application/json',
        },
    }

def lambda_handler(event, context):
    '''Demonstrates a simple HTTP endpoint using API Gateway. You have full
       access to the request and response payload, including headers
       and status code.'''
    print("Received event: " + json.dumps(event, indent=2))

    zip = event['queryStringParameters']['zip']
    print('ZIP -->' + zip)
    appid = event['queryStringParameters']['appid']
    print('Appid -->>' + appid)
    baseUrl = 'http://api.openweathermap.org/data/2.5/weather'
    completeUrl = baseUrl + '?zip=' + zip + '&appid=' + appid
    print('Request URL--> ' + completeUrl)

    req = Request(completeUrl)
    try:
        apiresponse = urlopen(completeUrl)
    except HTTPError as e:
```

```
        print('The server couldn\'t fulfill the request.')
        print('Error code: ', e.code)
        errorResponse = '{Error:The server couldn\'t fulfill the request: ' +
        e.reason +'}'
        return respond(errorResponse, e.code)
    except URLError as e:
        print('We failed to reach a server.')
        print('Reason: ', e.reason)
        errorResponse = '{Error:We failed to reach a server: ' + e.reason +'}'
        return respond(e, e.code)
    else:
        headers = apiresponse.info()
        print('DATE :', headers['date'])
        print('HEADERS :')
        print('---------')
        print(headers)
        print('DATA :')
        print('---------')
        decodedresponse = apiresponse.read().decode('utf-8')
        print(decodedresponse)
        return respond(None, decodedresponse)
```

Настройка Amazon API Gateway

После всего сделанного перейдите в сервис *Amazon API Gateway* в *AWS Console* — и увидите созданный вами API (рис. 9.6).

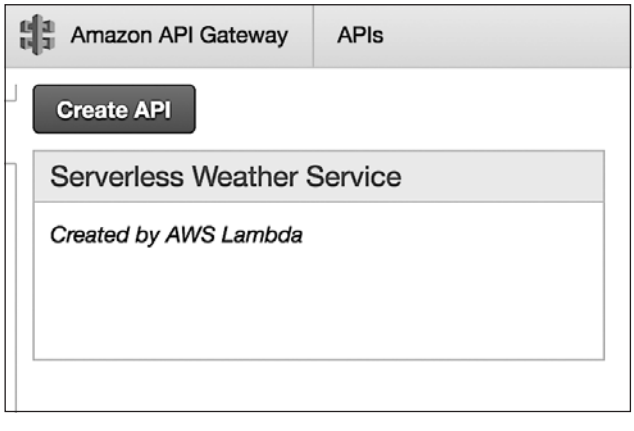


Рис. 9.6

Теперь настроим этот API под свои требования.

1. Щелкните на имени API Serverless Weather Service, чтобы перейти к его конфигурации.

2. Выберите слева раздел Resources и затем справа раскройте ресурсы, чтобы увидеть `/WeatherFunction/ANY`. Там должны быть подробности, приведенные на рис. 9.7.

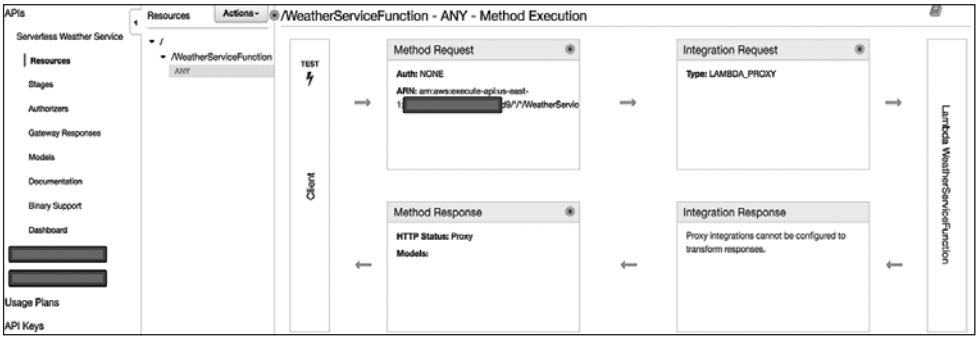


Рис. 9.7

3. Щелкните на Method Request и раскройте URL Query String Parameters. Воспользуйтесь кнопкой Add query string, расположенной снизу, чтобы добавить следующие параметры:
   - введите в поле Name значение `zip` и установите флажок Required;
   - добавьте еще один параметр, назначьте ему имя `appid` и также установите флажок Required.
4. На предыдущей панели выберите Request Validator и назначьте ему значение Validate body, query string parameters, and headers.
5. По завершении ваш экран Method Execution будет выглядеть как на рис. 9.8.

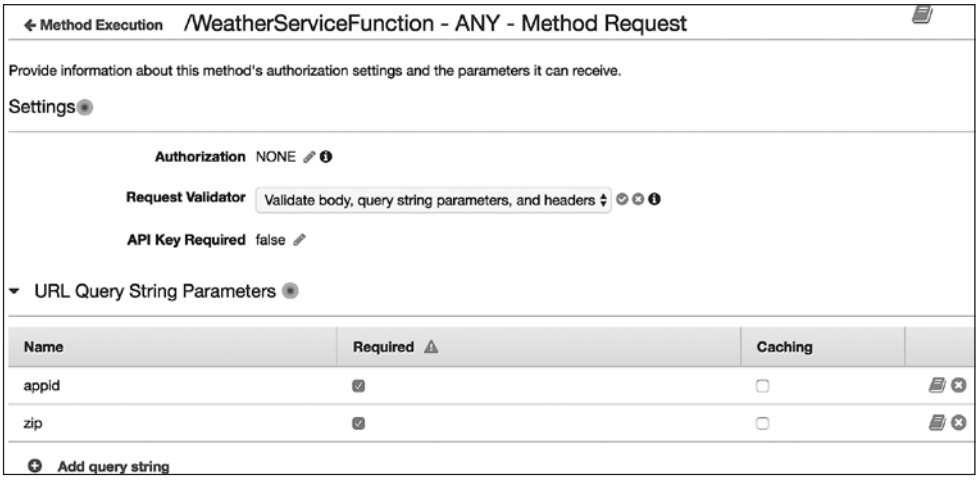


Рис. 9.8

Настройка аккаунта на сервисе OpenWeatherMap

Чтобы получать последние данные о погоде, наш микросервис будет вызывать API стороннего сервиса OpenWeatherMap (https://openweathermap.org/). Поэтому перейдите по ссылке и создайте новый бесплатный аккаунт. Сделав это, перейдите в настройки своего аккаунта и возьмите API-ключ на вкладке, изображенной на рис. 9.9.

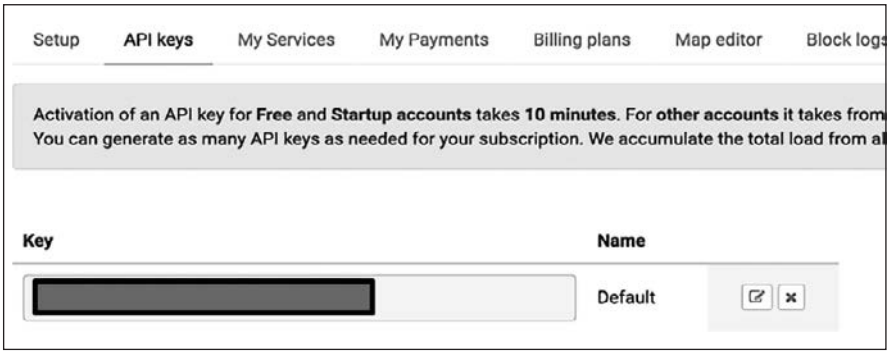


Рис. 9.9

Этот ключ будет использоваться для аутентификации ваших API-запросов к OpenWeatherMap.

Тестирование сервиса

Теперь, когда мы собрали все воедино, вернемся в Amazon API Console и протестируем сервис. Выполните следующие шаги.

1. Откройте файл `/WeatherServiceFunction/ANY` в Resources и выберите опцию Test.
2. Там выберите метод Get.
3. Для строки запроса нам понадобится два входных параметра: ZIP-код города, для которого вы хотите запросить данные о погоде, и API-ключ OpenWeatherMap. Для этого создайте строку запроса, например такую: `zip=<<zip-value>>&appid=<<app-id>>`. В этой строке замените конкретными значениями то, что выделено курсивом, и сформируйте строку в виде `zip=10001&appid=abcdefgh9876543qwerty`.
4. Нажмите Test внизу экрана, чтобы сделать вызов API и запустить функцию Lambda.
5. Если тест пройдет успешно, вы увидите соответствующий ответ со статусом Status: 200, а в теле ответа получите данные о погоде в формате JSON, как на рис. 9.10.

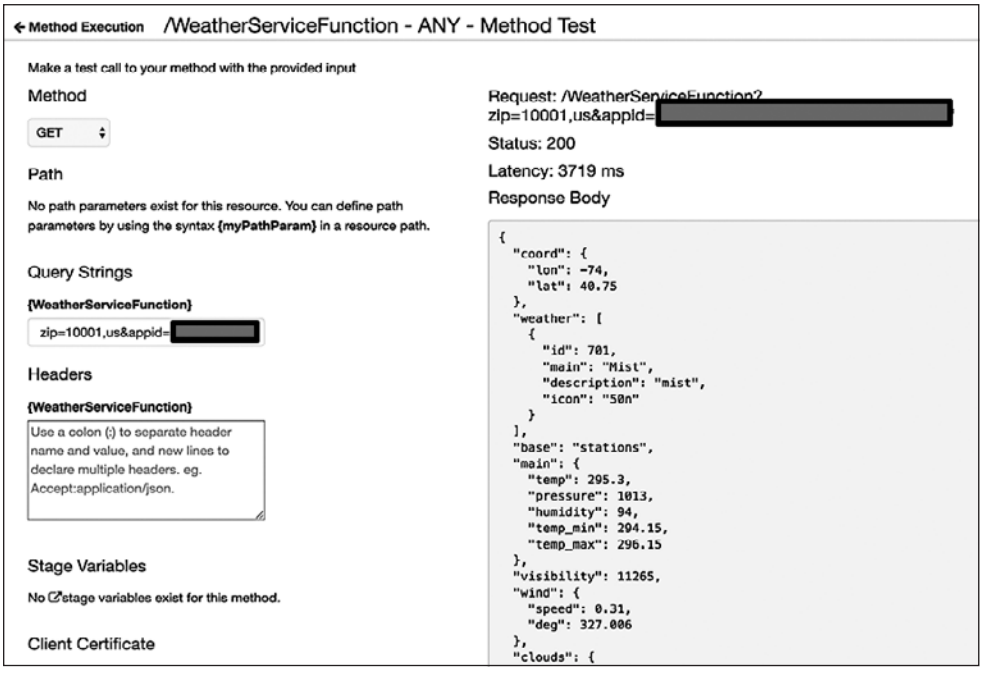


Рис. 9.10

Если тест провалится, проверьте журнальные записи в нижней части экрана, чтобы исправить проблемы.

Развертывание API

После успешного тестирования API нужно развернуть его, чтобы им можно было пользоваться в prod-окружении. Для этого выберите Deploy API в раскрывающемся меню Actions на экране конфигурации API (рис. 9.11).

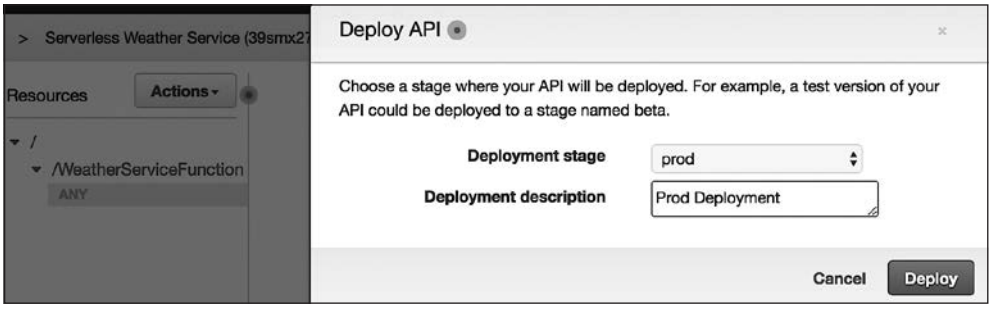


Рис. 9.11

Когда API будет успешно развернут, можете вернуться к Stages на левой панели, выбрать в категории Stage prod и под ним просмотреть метод GET, как на рис. 9.12.

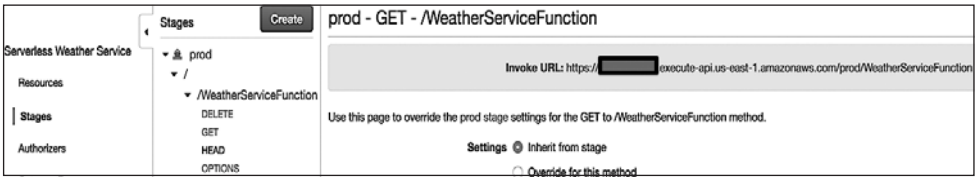


Рис. 9.12

Вы можете скопировать URL развернутого API и попробовать вызывать его, подставляя в качестве параметров значения `zip` и `appid`:

```
https://asdfgh.execute-api.us-east-1.amazonaws.com/prod/WeatherServiceFunct
ion?zip=10001,us&appid=qwertyasdfg98765zxcv
```

Замените выделенный текст значениями, специфичными для своего окружения.

Чтобы облегчить процесс тестирования развернутого API, можете воспользоваться HTML-шаблоном из репозитория https://github.com/PacktPublishing/Cloud-Native-Architectures.

1. Скачайте HTML-страничку, отредактируйте код для соответствия конечным точкам вашего API *на строке 28* и сохраните ее:

   ```
   <form method="get" action="https://<<обновленная-конечная-точка>>>/Prod/
   weatherservice">
   ```

2. Откройте страничку в браузере, заполните нужные поля и нажмите кнопку Get Weather, чтобы получить данные о погоде в данной локации (рис. 9.13).

Автоматизация бессерверных микросервисов с помощью AWS SAM

Как мы узнали из предыдущего раздела, бессерверный микросервис создать в консоли AWS очень легко. Но чтобы добиться того же самого с помощью автоматизированного подхода, следует задействовать другие механизмы. В среде AWS довольно популярно использование AWS CloudFormation, где вы можете создать собственные шаблоны для описания ресурсов AWS, а также их зависимости или параметры выполнения, необходимые для запуска приложения. Вы также можете визуализировать свои шаблоны в виде диаграмм и редактировать их с помощью

drag-and-drop-интерфейса или других компонентов. И, так как все, что вы делаете в CouldFormation, сохраняется в формате текстовых файлов JSON или YAML, вы сможете эффективно управлять своей инфраструктурой.

Рис. 9.13

С помощью CloudFormation можно создавать и обновлять AWS Lambda, API Gateway и множество других сервисов, которые могут быть составляющими ваших микросервисов. И несмотря на то, что это довольно легко, иногда приложение оказывается слишком простым для того, чтобы писать нечто длинное и продуманное для CloudFormation. В таких случаях рациональней пользоваться механизмами попроще. Для этого Amazon создали AWS *Serverless Application Model (SAM)*, который изначально поддерживается AWS CloudFormation и позволяет легко задавать API для Amazon API Gateway, функции для AWS Lambda и таблицы для Amazon DynamoDB, все необходимое для вашего бессерверного приложения.

AWS SAM — это мощный инструмент для быстрого создания бессерверных приложений, так как для него требуется намного меньше кода и конфигураций, чем для CloudFormation. Чтобы использовать AWS SAM для создания приложений, потребуются:

- YAML-шаблон для SAM с конфигурациями ваших ресурсов (функции Lambda, API Gateway и таблиц DynamoDB);
- код Lambda-функции в `.zip`-файле;
- API для API Gateway в формате Swagger.

Больше о AWS SAM вы можете прочитать в GitHub-репозитории https://github.com/aws/serverless-application-model/.

Теперь посмотрим, как создать такой же бессерверный сервис прогноза погоды с помощью AWS SAM.

YAML-шаблон для SAM

Далее рассмотрим пример YAML-шаблона для работы с Weather Service. По большому счету, он состоит из трех компонентов.

- API Gateway, который ссылается на файл Swagger для извлечения деталей конфигурации.
- Конфигурация функции AWS Lambda со ссылкой на код в ZIP-файле, именем обработчика и средой выполнения. Чтобы эту Lambda-функцию можно было связать с инстансом API Gateway, в ее конфигурации также есть перечень методов REST API и URL-путей, которые будут выступать ее триггерами.
- Последняя секция работает с выводом, который, в сущности, представляет собой URI конечной точки API стека, созданного SAM:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A simple cloud-native Microservice sample, including an AWS
SAM template with API defined in an external Swagger file along with Lambda
integrations and CORS configurations

Resources:
  ApiGatewayApi:
    Type: AWS::Serverless::Api
    Properties:
      DefinitionUri: ./cloud-native-api-swagger.yaml
      StageName: Prod

Variables:
        # NOTE: Before using this template, replace the <<region>>
        # and <<account>> fields
        # in Lambda integration URI in the swagger file to region
        # and account Id
        # you are deploying to
        LambdaFunctionName: !Ref LambdaFunction

  LambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./Cloudnative-weather-service.zip
      Handler: lambda_function.lambda_handler
      Runtime: python3.6
      Events:
        ProxyApiRoot:
          Type: Api
          Properties:
            RestApiId: !Ref ApiGatewayApi
            Path: /weatherservice
            Method: get
        ProxyApiGreedy:
```

```
            Type: Api
            Properties:
              RestApiId: !Ref ApiGatewayApi
              Path: /weatherservice
              Method: options
```

Вывод этого кода будет таким:

```
ApiUrl:
  Description: URL of your API endpoint
  Value: !Join
    - ''
    - - https://
        - !Ref ApiGatewayApi
        - '.execute-api.'
        - !Ref 'AWS::Region'
        - '.amazonaws.com/Prod'
        - '/weatherservice'
```

API в файле Swagger

Сегодня API используются повсеместно: в пакетных приложениях, облачных сервисах и др. И все же интеграция разных приложений может стать сущим кошмаром, если каждое из них использует собственные стандарты и механизмы определения и объявления своих API. Поэтому для такого рода стандартизации и структурирования мира REST API была создана инициатива OpenAPI. Подробнее можете почитать здесь: www.openapis.org.

Чтобы создать API, который будет соответствовать *OpenAPI Specification (OAS)*, Swagger (swagger.io) дает возможность использовать множество инструментов, которые будут помогать при разработке на протяжении всего жизненного цикла API, от проектирования и документирования до тестирования и развертывания.

AWS API Gateway поддерживает импорт и экспорт API формата Swagger, который основан на YAML. Есть несколько дополнительных элементов, которые можно добавить в файл Swagger для лучшей интеграции с AWS API Gateway. Далее приведен пример YAML для бессерверного микросервиса, в составе которого есть стандартные элементы Swagger и конфигурационные параметры для AWS API Gateway. Чтобы использовать этот пример в своей среде, пожалуйста, укажите в шаблоне свои параметры региона и аккаунта AWS из настроек AWS-аккаунта:

```
swagger: "2.0"
info:
  title: "Weather Service API"
basePath: "/prod"
schemes:
- "https"
paths:
  /weatherservice:
```

```
  get:
    produces:
    - "application/json"
    parameters:
    - name: "appid"
      in: "query"
      required: true
      type: "string"
    - name: "zip"
      in: "query"
      required: true
      type: "string"
    responses:
      200:
        description: "200 response"
        schema:
          $ref: "#/definitions/Empty"
    x-amazon-apigateway-request-validator: "Validate body, query string
    parameters,\
      \ and headers"
    x-amazon-apigateway-integration:
      responses:
        default:
          statusCode: "200"
      uri: arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
      functions/arn:aws:lambda:region:account number:function:
      ${stageVariables.LambdaFunctionName}/invocations
      passthroughBehavior: "when_no_match"
      httpMethod: "POST"
      contentHandling: "CONVERT_TO_TEXT"
      type: "aws_proxy"
  options:
    consumes:
    - "application/json"
    produces:
    - "application/json"
    responses:
    200:
      description: "200 response"
      schema:
        $ref: "#/definitions/Empty"
      headers:
        Access-Control-Allow-Origin:
          type: "string"
        Access-Control-Allow-Methods:
          type: "string"
        Access-Control-Allow-Headers:
          type: "string"
  x-amazon-apigateway-integration:
    responses:
      default:
```

```
          statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Methods:
              "'DELETE,GET,HEAD,OPTIONS,PATCH,POST,PUT'"
              method.response.header.Access-Control-Allow-Headers: "'Content-
              Type,Authorization,X-Amz-Date,X-Api-Key,X-Amz-Security-Token'"
              method.response.header.Access-Control-Allow-Origin: "'*'"
        requestTemplates:
          application/json: "{\"statusCode\": 200}"
        passthroughBehavior: "when_no_match"
        type: "mock"
definitions:
  Empty:
    type: "object"
    title: "Empty Schema"
x-amazon-apigateway-request-validators:
  Validate body, query string parameters, and headers:
    validateRequestParameters: true
    validateRequestBody: true
```

Код AWS Lambda

Код Lambda для этого приложения тот же, который использовался ранее. Вы можете скачать ZIP-файл с GitHub: github.com/PacktPublishing/Cloud-native-Architectures.

Использование AWS SAM

Под капотом AWS SAM задействует функциональность AWS CloudFormation (docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-updating-stacks-changesets.html). Чтобы создать бессерверный микросервис прогноза погоды с помощью SAM, сделайте следующее.

1. Скачайте YAML-шаблон для SAM, Swagger-файл для API Gateway и код для AWS Lambda в ZIP-файле и разместите их в одном каталоге.
2. Создайте корзину Amazon S3 (из консоли AWS или через API) и дайте ей любое доступное имя, например `CloudNative-WeatherService`.
3. Теперь нужно загрузить код Lambda и Swagger-файл в корзину S3, которую вы создали ранее, а затем добавить сведения о расположении S3 в YAML-файл для SAM. Чтобы это сделать, можете воспользоваться следующими консольными командами. Они все выполнят автоматически и создадут конечный YAML-файл для SAM, который можно будет применять в дальнейшем:

   ```
   aws Cloudformation package \
   --template-file ./Cloudnative-weather-microservice.yaml\
   --s3-bucket CloudNative-WeatherService \
   --output-template-file Cloudnative-weather-microservice-packaged.yaml
   ```

Если у вас все еще не установлен AWS CLI, можете руководствоваться документацией AWS: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html.

4. После успешного выполнения команды упаковки нужно начать развертывание. Для этого введите следующую команду, указав упакованный YAML-файл из предыдущего шага и имя создаваемого стека:

```
aws Cloudformation deploy \
--template-file ./Cloudnative-weather-microservice=-packaged.yaml\
--stack-name weather-service-stack \
--capabilities CAPABILITY_IAM
```

5. После успешного выполнения этой команды можете перейти в консоль AWS CloudFormation, чтобы посмотреть там статус стека. На вкладке Outputs выполненного стека CloudFormation вы увидите параметр под названием ApiURL со значением конечной точки API Gateway вашего Weather Service. Значит, API настроен и вы можете проверить это с помощью процедур, упомянутых в предыдущих разделах.

Автоматизация в AWS (ось 3 CNMM)

В Amazon долго поддерживалась традиция ведения малых команд, которые будут работать автономно и отвечать за выполнение всего цикла работ от планирования до обслуживания. Эти команды довольно гибкие, в них распределены роли (продакт-менеджер, разработчик, QA-инженер, инженеры по инфраструктуре/инструментам и т. д.) для того, чтобы управлять всеми аспектами жизненного цикла продукта, но в то же время они достаточно большие для того, чтобы можно было накормить их двумя пиццами вместо одной (рис. 9.14).

Рис. 9.14

Смысл существования команд двух пицц состоит в их автономности, динамичности и слаженной совместной работе, что позволяет сделать взаимодействие и рабочие процессы максимально эффективными. Это также создает идеальные условия с точки зрения DevOps, когда команда отвечает за весь жизненный цикл релиза, что включает в себя развертывание в среде эксплуатации и управление инфраструктурой.

Еще одним преимуществом этой идеи является то, что команды в итоге занимаются некой специфичной бизнес-функциональностью, которая часто интегрируется с другими компонентами системы посредством API на основе REST/HTTP. Это, по сути, основная концепция микросервисов, которая способствует достижению гибкости разработки и большей независимости компонентов для того, чтобы обеспечивать устойчивость и расширяемость всей системы.

В Amazon существует множество проектов и внутренних команд, работающих по такому принципу, они предоставляют всем расширяемую платформу для разработки под названием *Apollo*. Этот сервис для развертывания используют тысячи разработчиков в Amazon на протяжении многих лет, поэтому он хорошо проработан и вырос до платформы, соответствующей стандартам корпоративных продуктов.

Если вам интересна история Apollo, можете перейти в блог Вернера Фогельса (Werner Vogels), главного технического директора Amazon: https://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html.

Основываясь на опыте разработки внутренних проектов, в Amazon создали множество инструментов в рамках AWS. Они варьируются от простых API, SDK и поддержки CLI до полноценных сервисов для управления исходным кодом, непрерывной интеграцией и развертыванием, а также конвейерами разработки ПО. Подробнее об этом вы узнаете в следующем разделе.

Инфраструктура как код

Одни из основных преимуществ облака — возможность выстраивания автоматизированных процессов в инфраструктуре и отношение ко всем сущностям как к коду. Все сервисы AWS предоставляют свои интерфейсы REST API, а также поддерживают различные среды разработки для работы с популярными языками программирования, такими как Java, .NET, Python, Android и iOS, но один из самых мощных сервисов AWS — это AWS CloudFormation.

AWS CloudFormation позволяет разработчикам и системным администраторам с легкостью создать ряд связанных AWS-ресурсов и управлять ими, последовательно и предсказуемо обслуживая и обновляя их. Далее отображена схема-пример из

CloudFormation Designer, где вы можете увидеть инстанс EC2 с одним томом EBS и IP-адресом в Elastic (рис. 9.15).

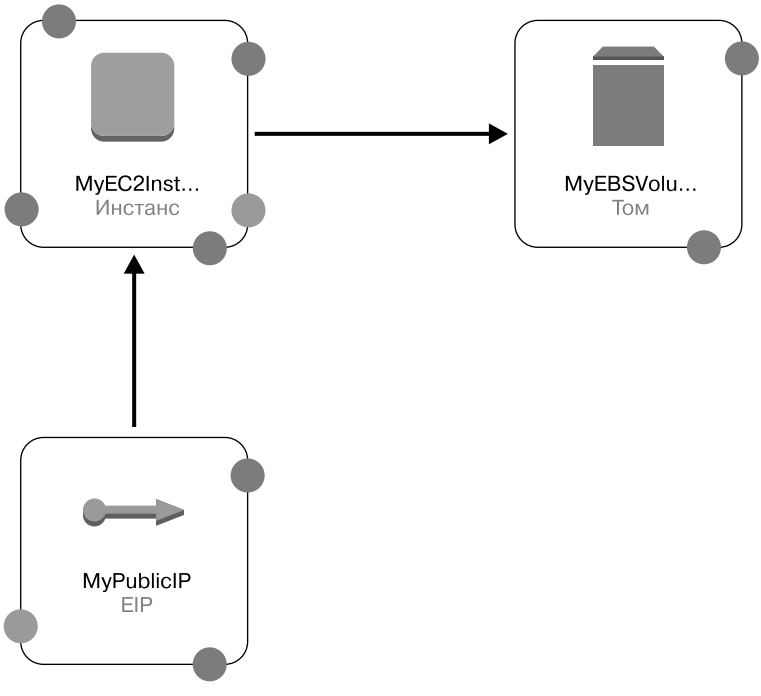


Рис. 9.15

Если превратить эту схему развертывания в YAML-шаблон для CloudFormation, он будет выглядеть так:

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  MyEC2Instance:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: ami-2f712546
      InstanceType: t2.micro
      Volumes:
        - VolumeId: !Ref EC2V7IM1
  MyPublicIP:
    Type: 'AWS::EC2::EIP'
    Properties:
      InstanceId: !Ref MyEC2Instance
  EIPAssociation:
```

```
    Type: 'AWS::EC2::EIPAssociation'
    Properties:
      AllocationId: !Ref EC2EIP567DU
      InstanceId: !Ref EC2I41BQT
  MyEBSVolume:
    Type: 'AWS::EC2::Volume'
    Properties:
      VolumeType: io1
      Iops: '200'
      DeleteOnTermination: 'false'
      VolumeSize: '20'
  EC2VA2D4YR:
    Type: 'AWS::EC2::VolumeAttachment'
    Properties:
      VolumeId: !Ref EC2V7IM1
      InstanceId: !Ref EC2I41BQT
  EC2VA20786:
    Type: 'AWS::EC2::VolumeAttachment'
    Properties:
      InstanceId: !Ref MyEC2Instance
      VolumeId: !Ref MyEBSVolume
```

Теперь, когда мы перенесли основные компоненты инфраструктуры в код, можно с легкостью управлять ими, как любым другим кодом приложения: включить в состав репозитория, отслеживать изменения, внесенные во время коммита, и просматривать перед развертыванием в промышленной среде. Эта концепция носит название *Infrastructure-as-Code (инфраструктура как код)*, она не только помогает с автоматизацией рутины, но и позволяет на более глубоких уровнях внедрять практики DevOps в любой среде.

Кроме сервисов AWS, существует множество популярных сторонних инструментов, таких как Chef, Puppet, Ansible, Terraform, которые помогают обслуживать компоненты инфраструктуры в манере *as code*.

CI/CD для приложений на Amazon EC2, Amazon Elastic Beanstalk

AWS предоставляет кучу сервисов для создания конвейеров непрерывной интеграции и развертывания для ваших приложений непосредственно в рамках AWS. Также существует множество инструментов, таких как Jenkins, Bamboo, TeamCity и Git, которые можно отдельно установить на AWS или даже интегрировать в сервисы непрерывной интеграции и непрерывной поставки (CI/CD), предлагаемые AWS. В целом, есть множество возможностей, среди которых пользователи смогут найти именно то, что им нужно.

Возможность	AWS-сервис	Другие инструменты
Репозиторий исходного кода, контроль версий, ветвление, управление тегами и т. д.	AWS CodeCommit	Git, Apache Subversion (SVN), Mercurial
Компиляция исходного кода и создание программных пакетов, готовых к развертыванию	AWS CodeBuild	Jenkins. CloudBees, Solano CI, TeamCity
Автоматизация тестов, охватывающих функциональность, безопасность, производительность или соответствие стандартам	Нет сервисов AWS, напрямую работающих с этим, но AWS CodeBuild может интегрироваться с различными инструментами для тестирования	Apica, Blazemeter, Runscope, Ghost Inspector
Автоматизированное развертывание на любом инстансе	AWS CodeDepoly	XebiaLabs

Кроме указанных сервисов, AWS предлагает два других сервиса, связанных с реализацией DevOps-практик.

AWS CodePipeline. Сервис помогает полностью моделировать и автоматизировать процессы релиза продукта в форме конвейеров. Конвейер определяет сценарий процесса релиза и описывает то, как новый код проходит через этот процесс. Конвейер состоит из ряда стадий (например, сборка, тестирование и развертывание), которые работают в качестве логических составляющих в вашем сценарии.

Далее приведен пример CI/CD-конвейера на CodePipeline, который использует различные сервисы, такие как AWS CodeCommit, AWS CodeBuild, AWS Lambda, Amazon SNS и AWS CodeDeploy, для управления всем процессом (рис. 9.16). Так как CodePipeline интегрируется с AWS Lambda, это дает дополнительные возможности по приведению действий и рабочих процессов в соответствие с требованиями ваших процедур CI/CD.

Выполняя упомянутый CI/CD-процесс, вы сможете пользоваться своевременными обновлениями в виде кода, развертываемого в одни и те же инстансы EC2, либо создавать новые инстансы для выполнения развертывания в стиле «синий/зеленый» (Blue/Green), где синий — это текущая среда, а зеленый — новая среда с последними изменениями кода. Концепция развертывания «синий/зеленый» основана на лучших облачных практиках *неизменяемой инфраструктуры*, где инстансы и окружение при необходимости заменяются полностью вместо того, чтобы частично обновляться с изменением некой конфигурации или установкой нового релиза. Эта идея аналогична неизменяемым переменным, присутствующим во многих практиках программирования, где вы их однажды инициализируете и больше никогда не обновляете, что позволяет делать процессы простыми и последовательными, избегая конфликтов из-за новых переменных или инстансов.

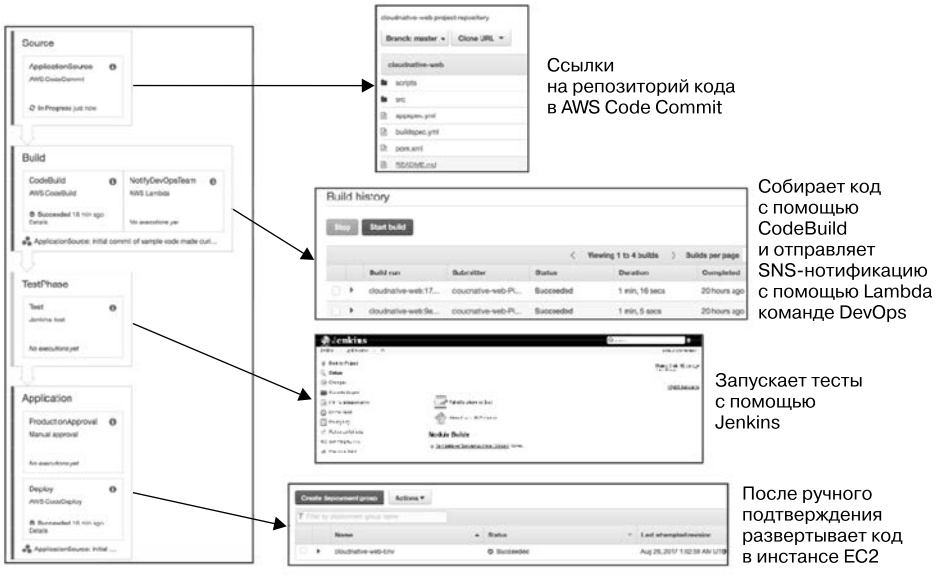


Рис. 9.16

Главные преимущества использования концепции Blue/Green.

- Вы не трогаете существующую среду, поэтому обновления становятся менее рискованными.
- Вы можете постепенно перейти на новую среду (Green) и так же легко вернуться к предыдущей среде (Blue), если возникнут проблемы.
- Вы обращаетесь с инфраструктурой, как с кодом, разрабатывая весь процесс Blue/Green в скриптах и процессах развертывания приложения.

Далее представлен пример *Blue/Green*-развертывания с помощью AWS CodeDeploy, где ряд автомасштабируемых инстансов EC2 на ELB (среда *Blue*) заменяется новым рядом автомасштабируемых инстансов (среда *Green*) (рис. 9.17). Вы найдете больше подробностей о процессе в документации AWS: docs.aws.amazon.com/codedeploy/latest/userguide/welcome.html#welcome-deployment-overview-blue-green:

Чтобы можно было быстро начать это использовать и интегрировать различные сервисы для AWS DevOps, AWS предлагает еще один сервис под названием AWS CodeStar. Он предоставляет унифицированный пользовательский интерфейс и панель управления проектом, включающую в себя интегрированную возможность отслеживания дефектов, позволяя вам с легкостью управлять процессом разработки с помощью одного и того же инструмента. Этот сервис предлагает множество шаблонов приложений, которые можно развернуть на Amazon EC2, AWS Elastic Beanstalk и AWS Lambda и обслуживать из различных DevOps-сервисов.

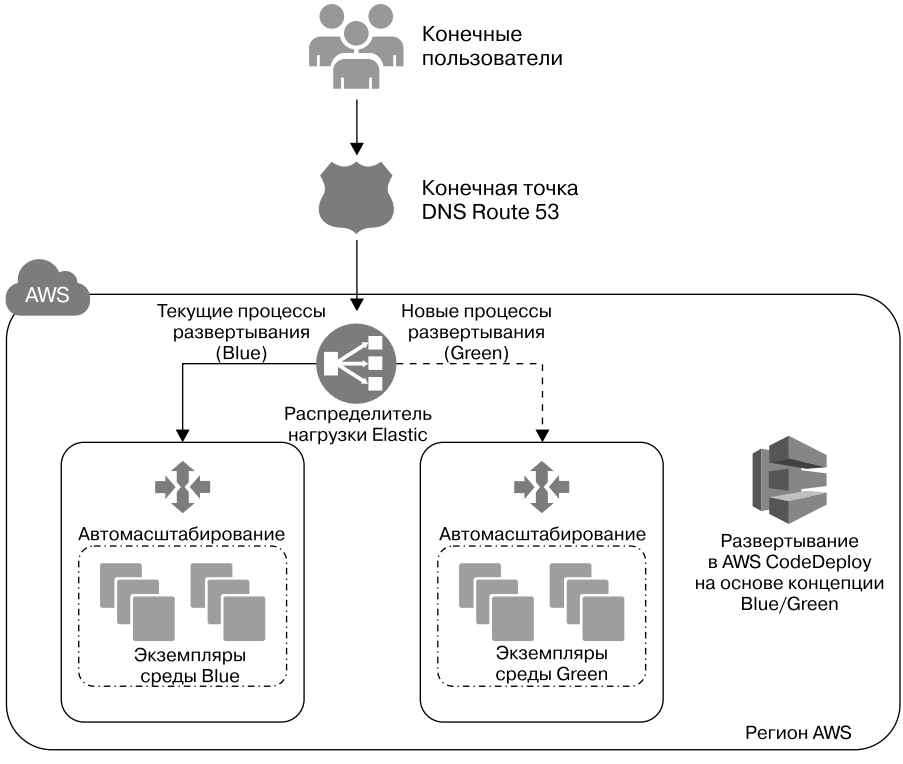


Рис. 9.17

На рис. 9.18 представлен снимок панели управления AWS CodeStar для одного из таких приложений.

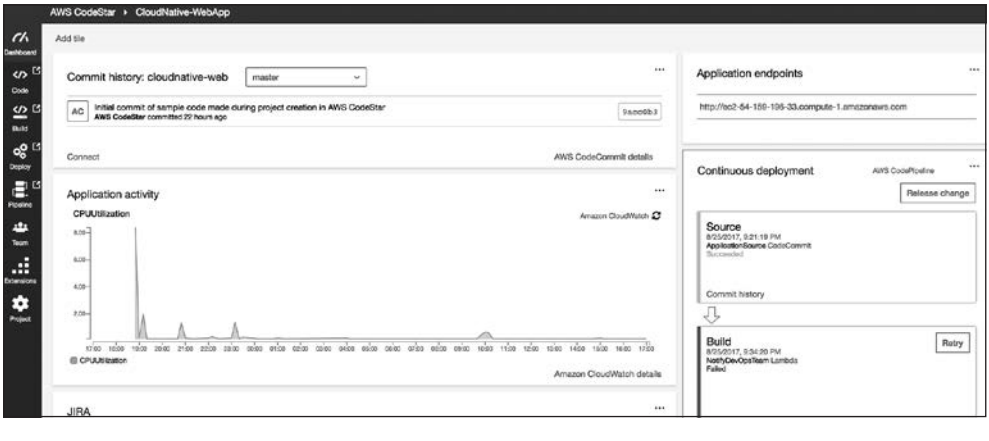


Рис. 9.18

CI/CD для бессерверных приложений

Процесс управления CI/CD-конвейерами весьма схож с процессами, описанными в предыдущем разделе об управлении средами приложения в Amazon EC2. Основные шаги, связанные с автоматизацией этого процесса для бессерверных приложений, представлены на рис. 9.19.

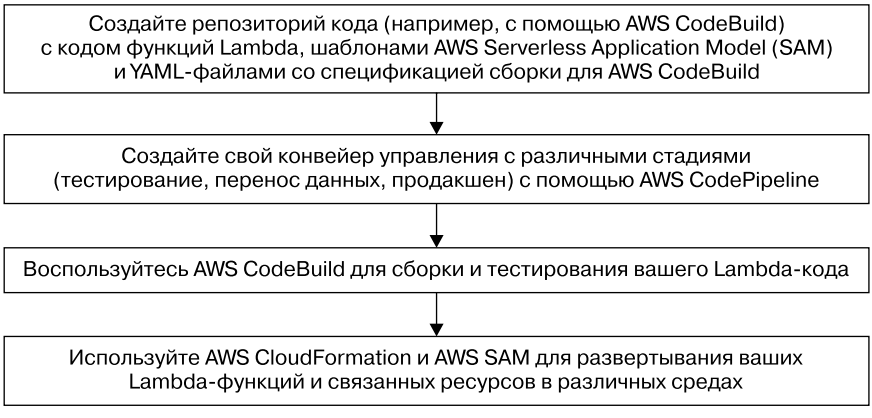
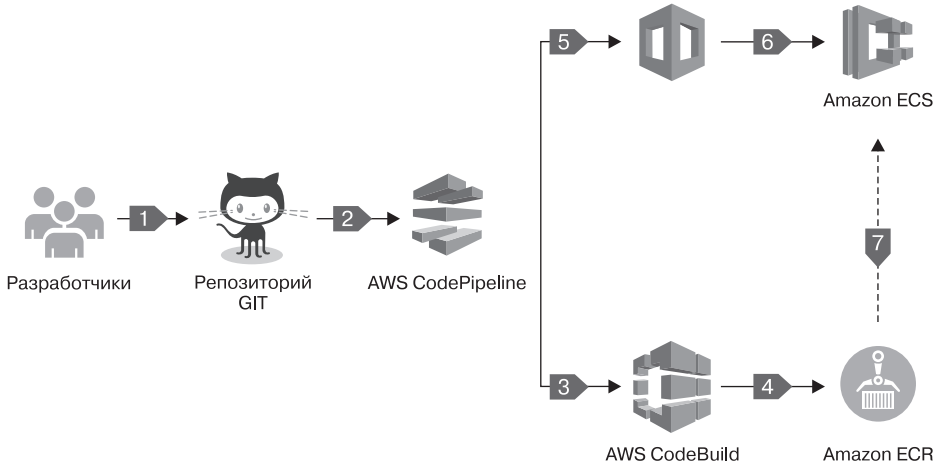


Рис. 9.19

CI/CD для Amazon ECS (Docker-контейнеры)

Контейнеры изменили мир упаковки программных продуктов, позволив создавать облегченные, независимые, исполняемые пакеты программного обеспечения, которые содержат все необходимое: код, среду исполнения, системные инструменты, системные библиотеки, а также настройки, обеспечивающие развертывание в различных вычислительных средах. Существует множество типов платформ, работающих с контейнерами, но одной из самых популярных остается Docker. Amazon предоставляют Amazon *EC2 Container Service (ECS)* и Amazon *Elastic Container Service for Kubernetes (EKS)* для надежного развертывания и обслуживания Docker-контейнеров в масштабных проектах. Amazon ECS и Amazon EKS интегрируются со множеством других сервисов AWS, таких как Elastic Load Balancing, тома EBS и роли IAM, чтобы процесс развертывания контейнерных приложений проходил еще легче. Вместе с Amazon ECS и Amazon EKS AWS предоставляют также Amazon *EC2 Container Registry (ECR)* для хранения, обслуживания и развертывания образов Docker-контейнеров в среде на Amazon ECS.

Для выполнения сборки CI/CD-сценария для развертывания различных версий приложений на Amazon ECS изучите схему использования сервисов, таких как AWS CodePipeline, AWS CodeBuild, AWS CloudFormation и Amazon ECR (рис. 9.20).



1. Разработчики непрерывно интегрируют свои изменения вместе в основную ветку, размещенную в исходном коде системы репозиториев, такой как GitHub.
2. AWS CodePipline берет код из репозитория и при обнаружении новой версии запускает выполнение конвейера непрерывной поставки.
3. AWS CodePipeline отправляет новую версию в AWS CodeBuild, который собирает образ Docker-контейнера из исходного кода.
4. AWS CodeBuild отправляет новую сборку образа Docker-контейнера, обозначенную идентификатором сборки, в репозиторий Amazon ECR.
5. AWS CodePipileine начинает обновление стека AWS CloudFormation, который обращается к Amazon ECS для назначения задачи и сервиса.
6. AWS CloudFormation создает новое определение задачи, ссылаясь на новую сборку образа, и обновляет сервис AmazonECS.
7. AmazonECS получает новый контейнер от AmazonECR и заменяет старую задачу на новую, чем и завершается процесс развертывания.

amazon web services™

Рис. 9.20

CI/CD для сервисов безопасности: DevSecOps

Как обсуждалось в предыдущем разделе, AWS имеет множество облачных AWS-сервисов и большинство из них включают в себя SDK и API, которые облегчают их интеграцию с различными приложениями и процессами. Еще одним ключевым аспектом является DevOps; это фактически сочетание культуры, философии, практических приемов и инструментов, улучшающих способность организации быстро развертывать приложения и сервисы. И теперь, когда приложения с невероятной скоростью поставляются в различные окружения, а проверки политики безопасности все еще не интегрированы в эти процессы, жутким кошмаром для организации становится то, что некоторые точки управления могут остаться без внимания и ими могут воспользоваться злоумышленники. На сегодняшний день общепринятое правило — предусмотреть безопасность в DevOps (его называют также DevSecOps или SecDevOps).

В основном этого добиваются тем, что обходятся с компонентами безопасности как с кодом, и интегрируют их в существующие практики DevOps. Например, если команда по безопасности определила, по каким стандартам VPC будут устанавливаться в любой среде, какие NACL-правила должны применяться и как следует создавать IAM-пользователей и их политики, эти нормы можно задокументировать в шаблонах CloudFormation, а команды приложения будут ссылаться на них и работать с ними, пользуясь возможностями вложенных стеков CloudFormation. Таким образом, команда по безопасности сосредоточена на создании и обновлении компонентов для безопасности и работы сетей — области своей ответственности, а команда приложения — на коде, процессах тестирования и развертывания на ряд серверов в соответствии с окружением.

Помимо применения при обслуживании инфраструктуры, эту практику можно вывести и на следующий уровень, внедряя проверки безопасности по всему конвейеру разработки и включая тестирование областей из следующего списка.

- Обеспечение интеграции инструментов анализа кода и анализа результатов, связанных с выводами о безопасности, перед запуском в производство.
- Включение журналов аудита с использованием таких сервисов, как AWS CloudTrail, помимо стандартного журналирования в приложении.
- Выполнение различных тестов безопасности, включая сканирование на вирусы и вредоносные программы, а также тестов на проникновение, которые могут потребоваться для рабочих нагрузок, обязанных соответствовать стандартам.
- Проверка баз данных и различных статических конфигураций и бинарных файлов, а также проверка того, что PHI/PII, такие как данные кредитных карт, номера социального страхования и сведения о здоровье, либо зашифрованы, либо замаскированы и недоступны в текстовом виде на случай всевозможных взломов.

Многие клиенты пользуются логикой и приложениями для автоматизации различных стадий конвейера, самостоятельно созданными с помощью AWS Lambda. Эти функции можно не просто исполнять напрямую, но еще и запускать с помощью, например, триггеров событий из таких сервисов, как CloudWatch monitoring, CloudWatch Events и SNS alerts.

AWS Config и Amazon Inspertor — еще пара сервисов, которые поспособствуют эффективной реализации DevSecOps. AWS Config поможет получать доступ, проверять и оценивать конфигурации ваших ресурсов AWS, а с Amazon Inspector вы можете автоматически оценивать приложения на уязвимости или искать расхождения с принятыми практиками, тем самым изначально предоставляя базовые правила и фреймворки по безопасности, которые можно дополнить вашими собственными политиками и проверками с использованием определений на основе AWS Lambda.

Методы перехода от монолитной архитектуры приложения к облачным архитектурам AWS

Ранее мы обсуждали множество вариантов создания облачных приложений с нуля с помощью различных сервисов AWS. Это было легко, так как можно все делать с начала, использовать различные облачные сервисы и создавать архитектуру своих приложений. Тем не менее если у вас уже имеется большой технический долг в среде эксплуатации и вы хотите уменьшить его, переместившись на облачную платформу, такую как AWS, то потребуется приложить больше усилий и лучше планировать.

В последние годы AWS заметно развила свои технологии и сервисы для миграции и передачи сообщений, чтобы упростить для организаций не только создание новых приложений, но и планирование реализации существующих на платформе AWS. У них есть технология *6Rs*, которая, возможно, охватывает все возможные сценарии миграции любого рода рабочей нагрузки на платформу AWS. 6Rs в AWS — это *Rehosting* (переход на новые серверы), *Replatforming* (перенос на новую платформу), *Repurshasing* (переприобретение), *Refactoring/ReArchitecting* (рефакторинг/перепроектирование), *Retire* (отстранение) и *Retain* (удержание). На рис. 9.21 представлена схема этой технологии.

Для разработки партнерской экосистемы с помощью технологии 6Rs AWS создали *Migration Acceleration Program (MAP)*, которая на данный момент является ключевым составляющим всего корпоративного сегмента стратегии миграции. Подробнее об этом вы сможете узнать здесь: aws.amazon.com/ru/migration-acceleration-program/.

Помимо этого, у AWS есть и более целостная платформа, предназначенная для любой организации, которая планирует переходить в облако, — AWS *Cloud Adoption Framework (CAF)*. Она охватывает шесть различных аспектов любой организации, таких как:

- бизнес;
- люди;
- управление;
- платформа;
- безопасность;
- администрирование.

Подробнее об AWS CAF можно узнать, перейдя по ссылке aws.amazon.com/professional-services/CAF/.

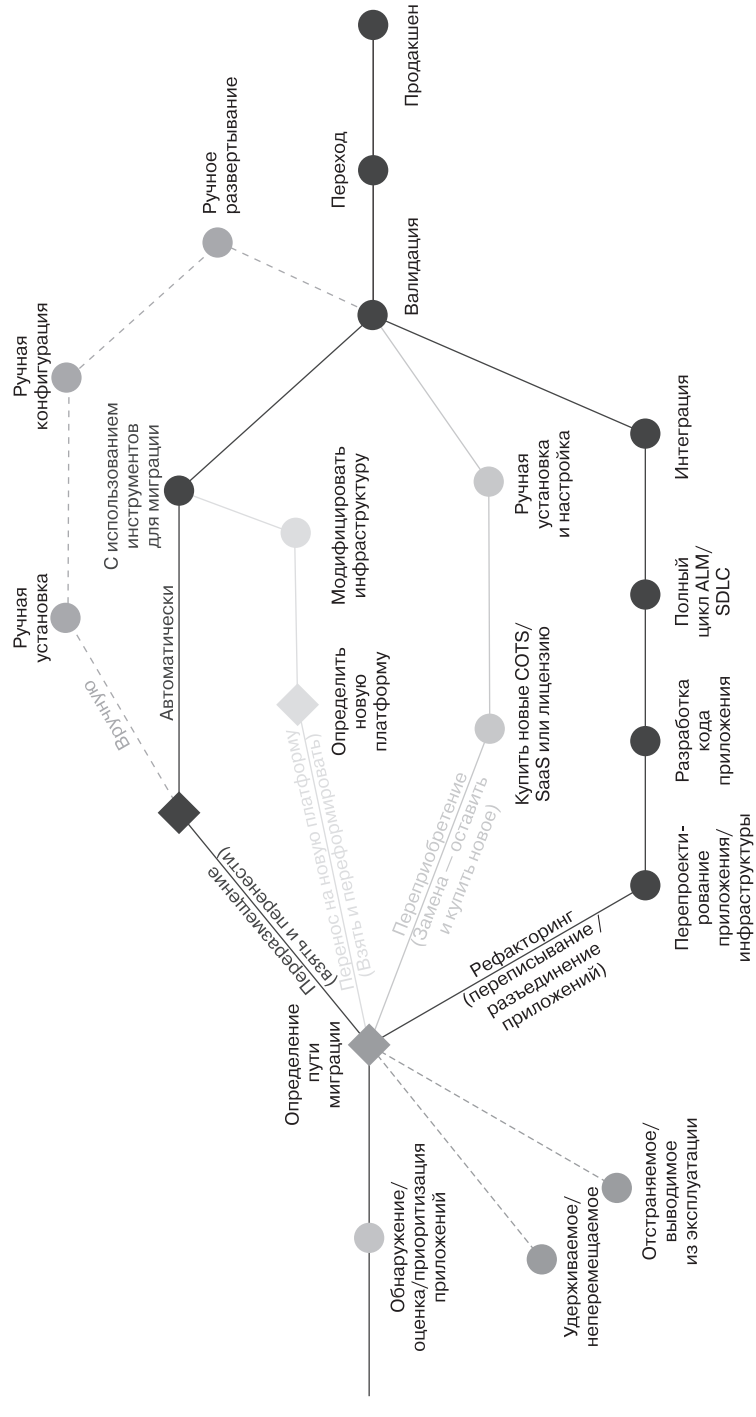

Продакшен
Переход
Валидация
Ручное развертывание
Ручная конфигурация
Ручная установка
Вручную
С использованием инструментов для миграции
Автоматически
Модифицировать инфраструктуру
Определить новую платформу
Ручная установка и настройка
Купить новые COTS/ SaaS или лицензию
Интеграция
Полный цикл ALM/ SDLC
Разработка кода приложения
Перепроекти-рование приложения/ инфраструктуры приложения
Переприобретение (Замена — оставить и купить новое)
Перенос на новую платформу (Взять и перефорфировать)
Рефакторинг (переписывание / разъединение приложений)
Перераспределение (взять и перенести)
Определение пути миграции
Обнаружение/ оценка/приоритизация приложений
Удерживаемое/ неперемещаемое
Отстраняемое/ выводимое из эксплуатации

Рис. 9.21

Резюме

В этой главе мы рассмотрели AWS и ее возможности в рамках CNMM, упомянутой ранее. Сосредоточившись на этом, изучили основные возможности AWS, направленные на создание облачных архитектур и эффективное администрирование приложений в этой среде. Затем рассматривали образцы бессерверных микросервисных приложений, оценивая различные ключевые сервисы AWS и фреймворки. И наконец, мы разобрались с принципами DevOps и DevSecOps, а также с тем, как они применяются в различных архитектурах приложений, начиная с развертывания на Amazon EC2 и заканчивая развертыванием на Amazon EC2 Container Service. Рассмотрели также шаблоны миграции приложений на AWS и концепцию 6Rs с фреймворком AWS CAF.

10 Microsoft Azure

История создания Microsoft Azure отличается от истории AWS. Изначально она была запущена как платформа Windows Azure, предназначенная для разработчиков и пользователей *Platform-as-a-Service (PaaS)*. Первые запущенные сервисы включали в себя Windows AppServer Fabric и ASP.NET MVC2, которые помогали разработчикам создавать приложения с помощью облачных платформ и сервисов. Из инфраструктурных компонентов на тот момент существовали только виртуальные машины Windows Server, рассчитанные на гибридные сценарии использования, однако общая стратегия была ориентирована на сообщество разработчиков, и это было совсем не так, как в AWS, где на то время предлагалось множество инфраструктурных компонентов, не говоря уже о некоторых базовых сервисах для приложений. С такими начинаниями в облаке Azure продолжали развиваться в области обслуживания приложений до 2014 года, а затем изменили стратегию и перешли в пространство IaaS, попутно совершив ребрендинг платформы в Microsoft Azure.

Взгляните на пост в блоге, сообщающий о запуске Windows Azure: https://news.microsoft.com/2009/11/17/microsoft-cloud-services-vision-becomes-reality-with-launch-of-windows-azure-platform/.

С того времени как Microsoft, запустив в июне 2012 года Azure Virtual Machines (широкодоступной она стала в апреле 2013-го), вышла на рынок облачных IaaS, Azure резко увеличила свое глобальное присутствие, а также количество предлагаемых услуг. На время написания этой книги Microsoft Azure охватила 36 регионов по всему миру и планирует в ближайшем будущем войти еще в шесть регионов. Стоит отметить незаметную на первый взгляд разницу в описании регионов у AWS и Microsoft Azure. Для AWS регион — это кластер зон доступности (AZ), каждая из которых может включать в себя один или несколько центров обработки данных, тогда как для Microsoft Azure регион — это географическая область, в которой есть центр обработки данных. Microsoft недавно объявила, что она также будет использовать AZ, как и AWS, но эта возможность только тестируется и потребуется время для того, чтобы она стала доступной повсеместно. Так что для пользователей, которым в это время нужна высокая доступность развертывания, в Microsoft Azure существует другая возможность под названием Availability Sets (подробнее

здесь: https://docs.microsoft.com/en-us/azure/virtual-machines/manage-availability#configure-multiple-virtual-machines-in-an-availability-set-for-redundancy).

Еще одна особенность облачных сервисов Microsoft, которую стоит упомянуть, — это то, что, помимо Microsoft Azure, предлагаются также сервисы из ряда Microsoft Office 365 и Microsoft Dynamics 365 CRM.

Чтобы быть в курсе последних новостей о Microsoft Azure и запусках новейших сервисов, подпишитесь на следующие ресурсы:

- обновления Azure: https://azure.microsoft.com/ru-ru/updates/;
- блог Microsoft Azure: https://azure.microsoft.com/ru-ru/blog/.

В этой главе мы рассмотрим следующие темы.

- Нативные облачные сервисы Microsoft Azure, их преимущества, место в циклах CI/CD, бессерверные сущности, контейнеры и концепции микросервисов, среди которых будут раскрываться:
  - функции Azure;
  - сервисы Visual Studio для командной работы;
  - сервис контейнеров Azure;
  - Azure IoT;
  - Azure Machine Learning Studio;
  - Office 365.
- Нативные облачные возможности работы с базами данных:
  - Azure Cosmos DB.
- Средства управления и мониторинга для нативных архитектур приложений на Microsoft Azure.
- Примеры архитектур и код для реализации CI/CD, бессерверных, контейнерных и микросервисных архитектур.

Облачные сервисы Azure (ось 1 CNMM)

Как обсуждалось в главе 1, первое и самое важное, что нужно в работе с облачными технологиями, — это понимание сущности сервисов поставщиков облака и их использования, так как это поможет сориентироваться не только в основных доступных уровнях инфраструктуры.

Согласно отчетам независимых аналитиков, Azure — единственный облачный провайдер, близкий к AWS с точки зрения облачных сервисов и возможностей выполнения. Такой статус достигается не только за счет того, что возможности

Azure охватывают ключевые области, такие как вычисления, хранение, сети и базы данных, но еще и благодаря множеству высокоуровневых сервисов для работы с данными, аналитикой, AI, Cognitive Services (https://azure.microsoft.com/ru-ru/services/cognitive-services/#overview), IoT, веб- и мобильными платформами и интеграцией на корпоративном уровне. В следующем подразделе мы рассмотрим некоторые из этих приложений и управляемых сервисов, которые могут помочь вам начать свободнее пользоваться облачными технологиями и оценить все преимущества данной платформы.

Платформа Microsoft Azure: ключевые инструменты

Как и в предыдущей главе об AWS, где мы рассматривали некоторые из основных сервисов Amazon, давайте взглянем на основные сервисы Azure, которые рекомендуется использовать для создания эффективных архитектур облачных приложений. Эти сервисы значительно увеличивают продуктивность команд разработчиков, ведь те могут сосредоточиться на бизнес-логике приложения, вместо того чтобы думать о том, как связать в одно целое ряд базовых возможностей, что зачастую может оказаться нерасширяемым и ненадежным решением по сравнению с тем, что способны предложить облачные провайдеры с их богатейшим опытом работы по всему миру. Ну что же, давайте взглянем на ключевые сервисы Azure, которые будут помогать в путешествии в мир нативных облачных технологий.

Azure IoT

Microsoft Azure предлагает ряд сервисов, которые помогают создавать связанные друг с другом платформы и решения, способные обрабатывать большие объемы данных, получаемых из множества устройств (и не только), на лету применяя к ним фильтры, анализируя их в реальном времени и храня в различных хранилищах для того, чтобы иметь возможность формировать различные показатели и необходимую функциональность. Один из основных сервисов, предназначенных для этого, — Microsoft Azure IoT Hub, который содержит множество функций, используемых для подсоединения устройства, управления его окружением и администрирования последнего. Далее представлены некоторые из основных функций этого обширного перечня.

- *Регистрация, аутентификация и авторизация устройства*. Один из важнейших моментов в области IoT — это обеспечение должного обслуживания и безопасного подсоединения устройства на бэкенде. Для этих целей Azure IoT предлагает такую возможность: с помощью портала и API можно создать отдельные сущности устройств, которые будут предоставлять их индивидуальные конечные точки в облаке. Этот метод имеет смысл, если количество используемых вами устройств ограниченно и вы можете добавить их все поочередно. Но если у вас тысячи устройств, которые нужно зарегистрировать, то в Azure IoT Hub есть отдельный метод `ImportDevicesAsync`, который можно применять

для одновременных загрузки, удаления или даже изменения статуса тысячи устройств вызовом одного-единственного API-метода. Таким же образом можно экспортировать информацию о зарегистрированных устройствах, но уже с помощью метода `ExportDevicesAsync`. После создания сущностей устройств будет создан симметричный ключ X.509, необходимый для аутентификации устройства. В комбинации со средствами управления доступом, сформированными согласно принятым политикам, и при наличии безопасного канала коммуникации (с TLS-шифрованием) все это позволит создать безопасную среду для установления соединений и обмена информацией с IoT Hub. На рис. 10.1 представлена схема этих средств управления (источник — ttps://docs.microsoft.com/ru-ru/azure/iot-fundamentals/iot-security-ground-up).

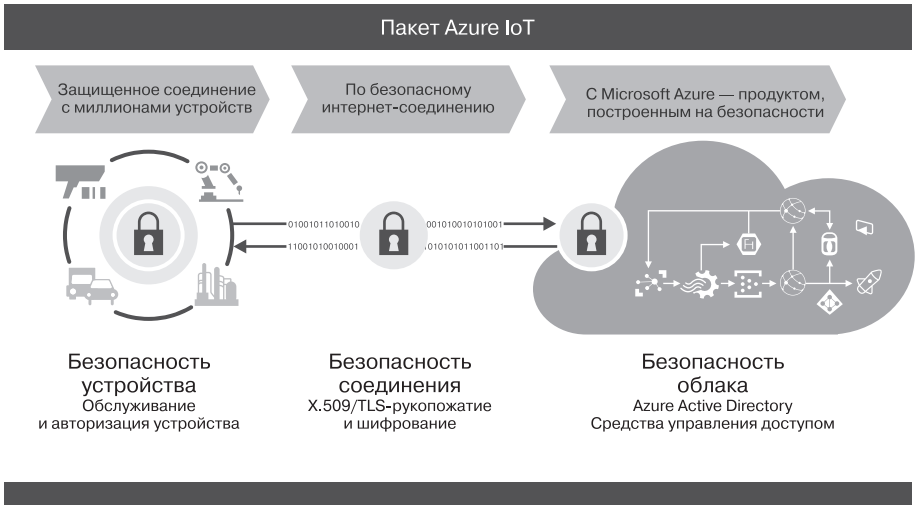


Рис. 10.1

- *Протоколы коммуникации*. Различные устройства используют разные типы протоколов и подходы к интеграции. Тем не менее в большинстве IoT-сценариев многие устройства поддерживают протоколы MQTT, AMQP и HTTPS. Azure IoT Hub также поддерживает все эти протоколы, тем самым обеспечивая сопряжение с большинством устройств без предварительных настроек. В то же время некоторые устройства поддерживают другие типы протоколов и механизмов коммуникации, что исключает их из ряда поддерживаемых по ранее упомянутым протоколам. Для таких случаев у Microsoft Azure IoT имеется Protocol Gateway (https://github.com/Azure/azure-iot-protocol-gateway/blob/master/README.md). Пользователи могут взять его с GitHub, внести нужные изменения и развернуть на инстансах Azure VM. Это упрощает обслуживание различных устройств, позволяя подключать их и управлять ими с помощью службы Azure IoT Hub (рис. 10.2, источник — docs.microsoft.com/ru-ru/azure/iot-hub/about-iot-hub).

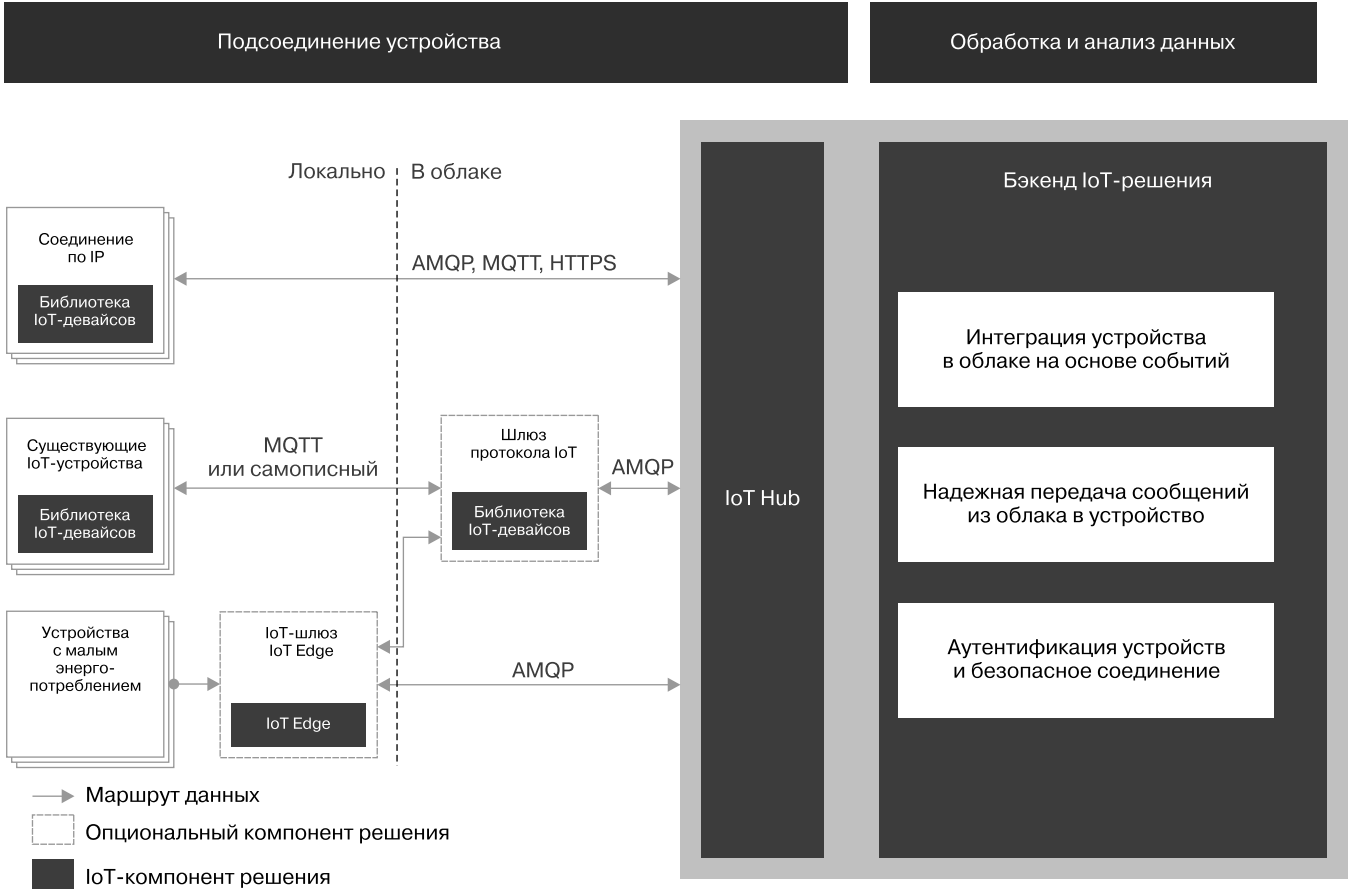

Подсоединение устройства
Обработка и анализ данных
Локально
В облаке
Соединение по IP
Библиотека IoT-девайсов
AMQP, MQTT, HTTPS
Существующие IoT-устройства
Библиотека IoT-девайсов
MQTT или самописный
Шлюз протокола IoT
Библиотека IoT-девайсов
AMQP
Устройства с малым энерго-потреблением
IoT-шлюз IoT Edge
IoT Edge
AMQP
IoT Hub
Бэкенд IoT-решения
Интеграция устройства в облаке на основе событий
Надежная передача сообщений из облака в устройство
Аутентификация устройств и безопасное соединение
Маршрут данных
Опциональный компонент решения
IoT-компонент решения

Рис. 10.2

- *Двойники устройств*. Устройства не всегда соединены с IoT Hub, но вашему бэкенд-приложению часто приходится запрашивать последний известный статус устройства или даже назначать ему определенный статус, как только оно присоединится к бэкенду с помощью IoT Hub, поэтому для операций по синхронизации статусов устройств Azure IoT Hub обеспечивает функциональность двойников устройств. Эти двойники представляют собой JSON-документы, у которых имеются теги (метаданные устройства) и свойства (желаемые или текущие). С их помощью вы можете создать нужную логику приложения для коммуникации с устройствами и их обновления (рис. 10.3, источник — docs.microsoft.com/ru-ru/azure/iot-hub/iot-hub-node-node-twin-getstarted).

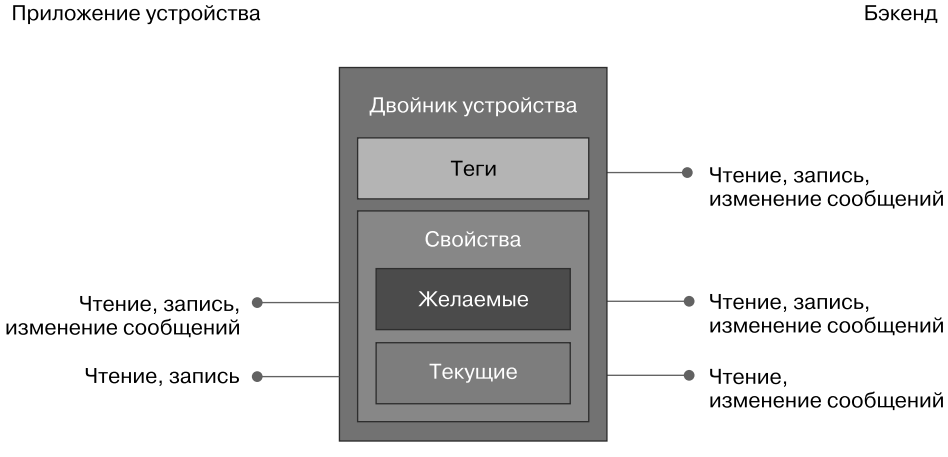


Рис. 10.3

- *Azure IoT Edge*. Иногда проще не отправлять данные из устройств в облако, а выполнить анализ или простые вычисления на периферийных компонентах. Типичным примером может служить ситуация, когда устройство располагается удаленно, а сетевое соединение слабое или дорогое, поэтому имеет смысл проводить анализ на стороне периферии. Для этого можно воспользоваться возможностью Azure под названием IoT Edge, когда код и вычисления запускаются ближе к самим устройствам с помощью множества SDK (C, Node.js, Java, Microsoft.NET и Python), предлагаемых Azure. Это позволяет выстроить гибридную облачную архитектуру, где некоторые вычисления будут выполняться на периферии, а более сложная обработка данных — в облаке, где различные сервисы помогут обеспечить необходимый масштаб обработки.

Помимо Azure IoT Hub, Azure предлагает готовые решения, которые клиенты могут развернуть одним щелчком кнопкой мыши и масштабировать до производственного уровня архитектуры в соответствии со своими требованиями. Эти решения используют множество сервисов Azure, таких как Azure IoT Hub, Azure

Events Hub, Azure Stream Analytics, Azure Machine Learning и Azure Storage. Это отличительная черта сервиса Azure IoT, так как другие поставщики облачных услуг предлагают только основные компоненты для обслуживания устройств и применения в IoT, в то время как Azure предоставляют готовые решения.

На рис. 10.4 представлен ряд решений, предлагаемых Azure IoT Suite на момент написания книги.

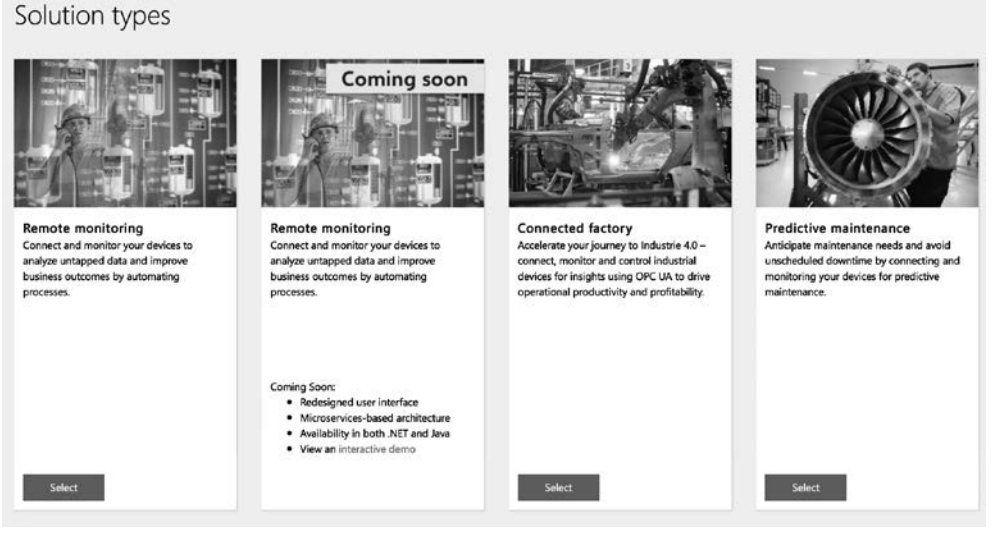


Рис. 10.4

Azure Cosmos DB

Azure Cosmos DB — это распространенная по всему миру база данных, работающая по множеству моделей. Azure Cosmos DB позволяет гибко и независимо масштабировать пропускную способность и объем хранилища в любых географических регионах Azure. Azure Cosmos DB с 99%-ной гарантией обеспечивает величину задержки в пределах нескольких миллисекунд в любой точке мира, предлагает множество проверенных моделей поддержания показателей производительности и обещает высокую доступность с поддержкой множественной адресации (подробности можно узнать здесь: azure.microsoft.com/ru-ru/services/cosmos-db/).

Azure Cosmos DB запущена в мае 2017 года, но она не является совершенно новым продуктом. Многое она унаследовала от предшественницы — запущенной на несколько лет раньше Azure DocumentDB, которая была больше сосредоточена на архитектурных шаблонах, основанных на NoSQL. В то же время Azure Cosmos DB — это не просто перевоплощение Azure DocumentDB с новым именем, в ней

появилось множество новых функций и возможностей, которых прежде не было. Далее представлены три важные особенности Cosmos DB.

- *Глобально распределенная модель развертывания*. В Azure есть множество регионов, и если вы хотите развернуть приложение, которое будет работать в нескольких из них с распределенной между ними базой данных, то Cosmos DB позволит реализовать эту возможность с помощью глобальной модели развертывания. Вы можете выбирать регионы, в которых хотите расположить копии инстансов DB, как во время создания базы данных, так и когда она уже начнет действовать. Также можете указать, какие регионы будут иметь права чтения и/или записи в соответствии с вашими требованиями. Вдобавок можете задать приоритеты резервных ресурсов для каждого региона на случай каких-то больших локальных событий или проблем.

  В некоторых случаях вам может понадобиться ограничить область использования данных до особенной локации или региона (например, для защиты согласованности, приватности данных или в рамках соблюдения внутригосударственных правил обращения с данными), и, чтобы добиться соответствия этим требованиям, вы можете задействовать политики, управляемые с помощью метаданных вашей подписки Azure (рис. 10.5).

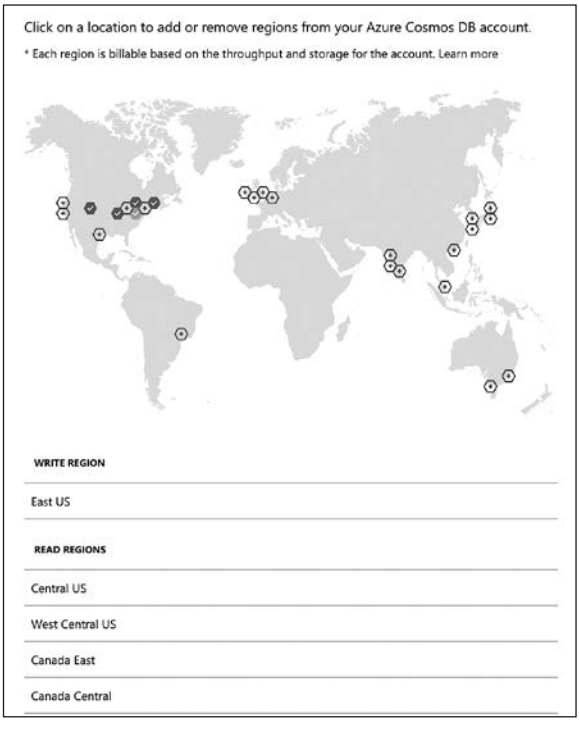


Рис. 10.5

- *Многомодельные API.* Это одна из отличительных особенностей по сравнению с более ранней версией Azure DocumentDB, которая на данный момент поддерживает множество других моделей.
  - *Графовая база данных* — это модель, в которой данные имеют множество общих точек и ребер. Каждая точка задает уникальный объект среди данных (например, человека или устройство), который может быть соединен с *n* других точек посредством ребер, задающих между ними связь. Чтобы делать запросы к Azure Cosmos DB, вы можете использовать язык обхода графа Apache TinkerPop, Gremlin или другие графовые системы, совместимые с Tinker-Pop, такие как Apache Spark GraphX.
  - *Таблица* — это модель, основанная на табличном хранилище Azure, что представляет собой, по сути, NoSQL-подход с ключами/значениями для хранения значительного количества частично структурированных наборов данных. Cosmos DB предоставляет возможность глобального распространения в дополнение к API табличного хранилища Azure.
  - *JSON-документы* — Cosmos DB предоставляет возможность хранить в вашей базе данных JSON-документы, на которые можно ссылаться с помощью существующих API от DocumentDB или новых API от MongoDB.
  - *SQL* — Cosmos DB поддерживает некоторые базовые функции SQL посредством существующих API от DocumentDB. Тем не менее это не полноценная SQL-база данных от Azure, и если требуется значительная поддержка операций SQL, то Cosmos DB может оказаться не совсем оптимальным выбором.
- *Модели согласованности.* Обычно поставщики обеспечивали не очень много возможностей для распространения данных в различных модулях и географических точках. Далее представлены самые распространенные варианты распределенных вычислений:
  - *строгая согласованность*, когда ответ на запрос возвращается только в том случае, если изменения были успешно зафиксированы во всех репликах, благодаря чему операция чтения, выполненная после записи, всегда гарантирует получение самого последнего значения;
  - *отложенная согласованность* (или согласованность в конечном счете), когда ответ после запроса возвращается незамедлительно и различные модули синхронизируются с новыми данными, из-за чего после чтения и записи может отображаться старое значение.

Однако Cosmos DB поднимает эту модель на новый уровень. Стали предоставлять и другие модели согласованности (ограниченное устаревание, сессия и последовательный префикс), которые являются чем-то средним между строгой и отложенной согласованностью (рис. 10.6).

На рис. 10.7 представлен снимок с портала Azure Cosmos DB, где показаны эти настройки моделей согласованности данных.

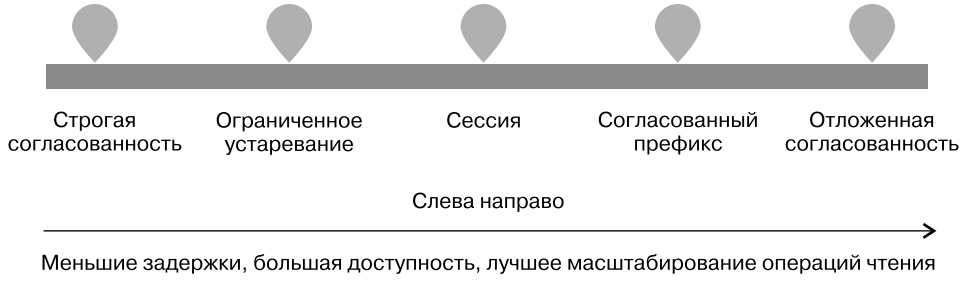


Рис. 10.6

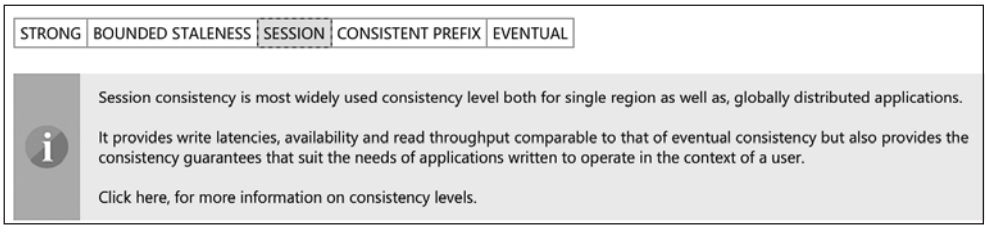


Рис. 10.7

И наконец, хочется сказать, что все эти возможности Azure Cosmos DB прекрасно подойдут для реализации множества сценариев с применением облака, таких как общие базы данных для глобально распределенных архитектур приложений, телеметрии и информации о состоянии для связанных платформ/IoT и обслуживания на стороне бэкенда для бессерверных приложений.

Azure Machine Learning Studio

Microsoft выпустила этот сервис как Azure Machine Learning, но позже название сменили на Azure Machine Learning Studio, чтобы оно соответствовало основным предоставляемым им возможностям, включающим в себя управление процессом использования самой Studio (Visual Workbench). Это управляемый сервис, который предназначен для создания, тестирования, администрирования и обслуживания аналитических решений в облаке. Основная его ценность состоит в том, что вам нет нужды быть специалистом по работе с данными, чтобы успешно создавать модели машинного обучения, так как он предоставляет множество пробных наборов данных и модулей анализа, которые вы можете использовать для создания среды для машинного обучения. Опробовав, протестировав и обучив свою экспериментальную сущность, вы сможете применять ее для начала анализа и прогнозирования на основе имеющихся данных, а также сделать доступной в качестве веб-сервиса. На рис. 10.8 представлена схема, которая подытоживает озвученные ранее концепции (источник — https://bit.ly/3f9QI1G).

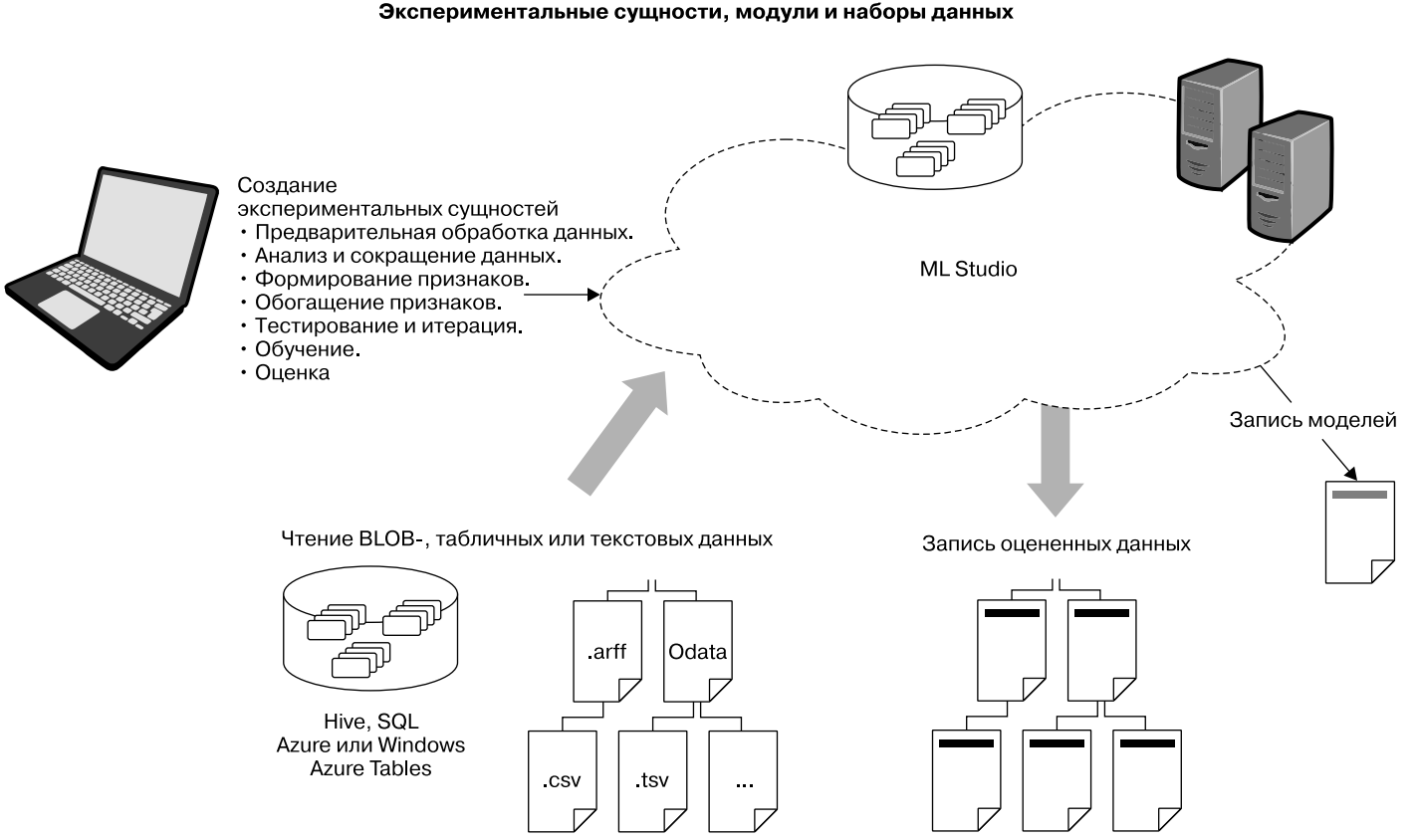

Экспериментальные сущности, модули и наборы данных
Создание
экспериментальных сущностей
• Предварительная обработка данных.
• Анализ и сокращение данных.
• Формирование признаков.
• Обогащение признаков.
• Тестирование и итерация.
• Обучение.
• Оценка
ML Studio
Запись моделей
Чтение BLOB-, табличных или текстовых данных
Hive, SQL
Azure или Windows
Azure Tables
.arff
Odata
.csv
.tsv
...
Запись оцененных данных

Рис. 10.8

Одно из важнейших преимуществ Aszure Machine Learning Studio заключается в том, что существует множество готовых к использованию алгоритмов.

- *Обнаружение аномалий*. Обнаруживает и предсказывает редкие или необычные точки данных, например прогнозирует риски по кредитам или выявляет вредоносную деятельность.
- *Классификация*. Определяет, к какой категории относится информация. Это могут быть простые категории типа «да/нет», «правда/ложь» или сложные, такие как анализ мнений в социальных сетях.
- *Кластеризация*. Определяет похожие точки данных в отдельные интуитивно понятные группы, например, по прогнозированию вкусов покупателей или степени заинтересованности в конкретном товаре.
- *Рекомендация*. Прогнозирует вероятность того, что товар придется по нраву пользователю, исходя из ранее зарегистрированных его действий. Типичное применение технологии — в онлайн-торговле, где на основе вашей истории запросов формируются дополнительные варианты товаров и услуг.
- *Регрессия*. Прогнозирует будущее, оценивая отношения между переменными, например прогнозирование объемов продаж автомобилей на пару месяцев вперед.
- *Статистические функции*. Математические вычисления значений из колонок базы данных, вычисление корреляций, *z*-оценка, оценка вероятности и т. д.
- *Текстовый анализ*. Работает со структурированным и неструктурированным текстом для определения языка вводимого текста, создает словари *n*-грамм и извлекает имена людей, названия мест и организаций из неструктурированного текста.
- *Компьютерное видение*. Обработка изображений и их распознавание для идентификации черт лица, разметки изображений, цветового анализа и т. д.

Чтобы больше узнать об алгоритмах Azure Machine Learning, перейдите по ссылке docs.microsoft.com/ru-ru/azure/machine-learning/algorithm-cheat-sheet.

Как видите, существует множество доступных алгоритмов, тем не менее разработчики/специалисты по работе с данными часто нуждаются в расширении или модификации некоторой функциональности, поэтому здесь можно использовать также скриптовые модули R или Python и Jupyter Notebook.

Как уже упоминалось ранее, выполняемый вами рабочий процесс машинного обучения можно сделать доступным для внешних приложений с помощью вебсервисов REST. Для этого существуют две основные возможности, связанные с типом выполнения моделей:

- *Request Response Service (RRS)* — для синхронного выполнения без запоминания состояния;
- *Batch Execution Service (BES)* — для пакетной асинхронной обработки.

Сервисы Visual Studio Team

Как говорилось в предыдущей главе, AWS предоставляет множество сервисов, позволяющих успешно реализовывать DevOps-практики в любой организации. Таким же образом Azure предлагает ряд сервисов/функций для DevOps, которые помогают разработчикам создавать приложения, работающие по принципам и процессам CI/CD. К тому же даже до того, как в Microsoft появились облачные сервисы, у них уже была штатная платформа Team Foundation Services (TFS), которую разработчики могли применять для создания и поставки качественных продуктов. После смены приоритетов в пользу облака Microsoft запустила облачный сервис Visual Studio Team Services (VSTS), который является *облачной* версией уже существовавшей TFS. Основное преимущество VSTS — конечно, то, что это управляемая платформа. Azure выполняет бóльшую часть рутинной работы от имени клиентов, чтобы они могли сосредоточиться на своих основных бизнес-целях: разрабатывали приложения, выстраивая эффективную коммуникацию и практики. А так как TFS к этому времени существовала уже довольно долго, VSTS стал многофункциональной платформой со множеством встроенных функций и возможностей сторонней интеграции.

Чтобы получше понять, в чем отличие VSTS от TFS[1], можете перейти по ссылке docs.microsoft.com/en-us/vsts/user-guide/about-vsts-tfs.

Поскольку мы здесь обсуждаем облачные возможности, давайте глубже погрузимся в изучение VSTS и обсудим основные из предоставляемых им функций.

- Первая и самая важная — это возможность обслуживать код, различные его версии и ветви. Для этих целей VSTS предлагает несколько вариантов проектирования инфраструктуры, которые включают в себя Git (распределенный) или Team Foundation Version Control (TFVC), и централизованную клиент-серверную систему. Git-репозитории разработчики могут задействовать с помощью разных IDE, включая Visual Studio, Exlipse, Xcode и IntelliJ. А для работы с TFVC вы можете применять Visual Studio, Eclipse и Xcode. Пользовательский интерфейс облачного VSTS также предлагает множество возможностей для непосредственного исследования файлов кода в проекте и просмотра подробностей push/pull-комитов.
- Вторая важная функция — это возможность внедрения непрерывной интеграции и непрерывного развертывания, которая является ключевой для успешной реализации DevOps-практик. И опять же VSTS предоставляет для этого множество функций, таких как возможность создания конфигураций для сборок

[1] В 2018 году Microsoft переименовала Visual Studio Team Services (VSTS) в Azure DevOps Services. Visual Studio Team Foundation Server переименован в Azure DevOps Server. Подробнее здесь: https://docs.microsoft.com/en-us/azure/devops/server/tfs-is-now-azure-devops-server. — *Примеч. пер.*

и релизов, библиотек переменных сборки, которые могут быть особыми в каждом из окружений, а также для создания групп развертывания, где можно задать целевые инстансы, включая Windows, Ubuntu или RHEL.

- Для того чтобы реализовывать практики DevOps, многие команды используют Scrum, включая технологии разработки по Kanban. Поэтому важно внедрять эти процессы в практики разработки кода, и это как раз то, с чем VSTS способен помочь, позволяя пользоваться панелями управления, бэклогами спринтов, панелями задач и связанными с ними инструментами визуализации для планирования и совместной работы. Это помогает проектным менеджерам не только понять общее состояние дел — им становится легче формулировать глобальные задачи, разработчики могут связывать их задачи/исправление дефектов/проверки с соответствующими задачами, а QA-команды могут описывать их тестовые сценарии и фиксировать результаты. Все вместе это дает возможность выстраивать системы, работающие в формате end-to-end.
- Кроме основных функций, важно еще иметь возможность непосредственно или без больших сложностей интегрироваться с партнерскими экосистемами. Это как раз то, что готов предложить Visual Studio Marketplace. Вы можете использовать множество плагинов, включая те, что работают с визуализацией рабочих процессов, поиском кода, интеграцией slack и т. п. Перейдите по ссылке https://marketplace.visualstudio.com/vs, чтобы подробнее узнать о доступных программных пакетах и возможностях интеграции.

Любопытно, что VS Marketplace предлагает также плагины для управления интеграцией сервисов AWS, таких как Amazon S3, AWS Elastic Beanstalk, AWS CodeDeploy, AWS Lambda и AWS CloudFormation: https://marketplace.visualstudio.com/items?itemName=AmazonWebServices.aws-vsts-tools.

Office 365

Одним из самых популярных продуктов Microsoft уже многие годы является Microsoft Office. На заре облачных технологий компания Microsoft начала предлагать его SaaS-версию под названием Office 365. Это позволяло любому (обычному пользователю, владельцу бизнеса или студенту) зайти в Интернет, приобрести подписку на Office 365 и начать работать с пакетом прямо из любимого браузера. Сегодня, хотя это и облачное решение, оно не поставляется как компонент Azure, так как Azure больше сосредоточен на инфраструктуре и сервисах платформы. Тем не менее мы рассмотрим Office 365, так как это важная составляющая облачного бизнеса Microsoft и для многих компаний, переходящих в облако, Office 365 также станет одним из важнейших приложений, на которое они будут переходить в процессе трансформации. На рис. 10.9 представлены приложения, предлагаемые Microsoft в составе Office 365.

Как говорилось ранее, между Azure и Office 365 существует тесная связь, и одна из основных областей интеграции — это аутентификация. Под этим подразумевается то, что Office 365 применяет Azure AD для управления сущностями пользователей. Еще одно преимущество этой связи заключается в том, что если клиенты штатно задействуют Active Directory для управления сущностями пользователей, то они могут синхронизировать тамошние пароли с Azure AD и установить единый вход. Конечным пользователям станет легче, им не придется запоминать дополнительные пароли, кроме паролей своих корпоративных пользовательских сущностей, и вход в Office 365 также будет интегрирован.

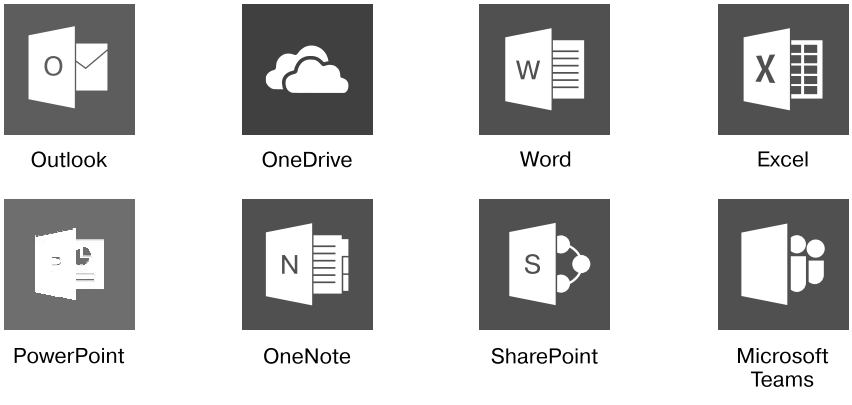


Рис. 10.9

Еще одной причиной важности Office 365 является то, что этот пакет конкурирует с продуктами других поставщиков облака. Так, несмотря на то, что AWS предлагает ряд бизнес-приложений (например, Amazon Workdocs и Amazon Chime), а у Google есть G Suite, с точки зрения больших корпораций Microsoft в этой области явно лидирует уже многие годы. Office де-факто был платформой для обеспечения совместной работы и повышения производительности. Клиентам легче решиться на переход в облако и интеграцию с Office 365, так как это минимально меняет их бизнес.

Проектирование, ориентированное на приложения (ось 2 CNMM)

Как обсуждалось в предыдущей главе, создание бессерверных и микросервисных приложений — это основное отличие облака от шаблонов проектирования дооблачных времен. Давайте посмотрим, как можно проектировать приложения в облаке Microsoft Azure с помощью ряда ключевых сервисов.

Бессерверные микросервисы

В этом разделе мы разработаем бессерверное микросервисное приложение в Microsoft Azure. Чтобы было легче сравнивать и учиться на примерах деятельности различных поставщиков облака, будем создавать все то же приложение Weather Services, о котором говорилось в предыдущей главе об AWS. Напомним, что приложение будет состоять из трех основных частей:

- API-триггера для вызова приложения;
- функции, написанной в Azure Functions;
- внешнего сервиса, который будет получать от нас некоторые параметры и возвращать результаты (рис. 10.10).

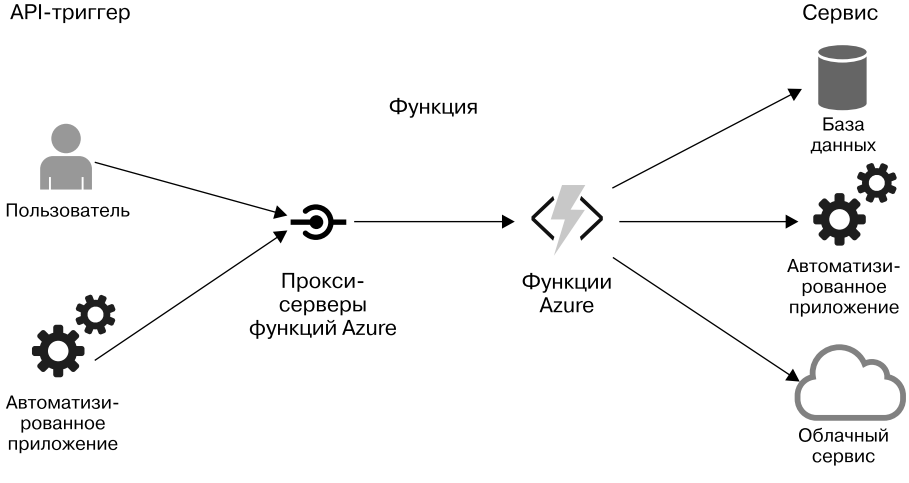


Рис. 10.10

Пример

Для создания упомянутого ранее приложения на сайте Azure сделайте следующее.

1. Нажмите кнопку Create a resource в верхнем левом углу, а затем выберите Compute | Functions App.
2. Создайте Functions App, задавая настройки, показанные на рис. 10.11 (некоторые из них могут отличаться от представленных и должны соответствовать параметрам вашего аккаунта и существующим ресурсам).
3. После этого нужно создать основную бизнес-логику в форме функции Azure. Поэтому раскройте новое приложение и нажмите кнопку + рядом с Functions, чтобы создать функцию HTTP trigger (рис. 10.12).

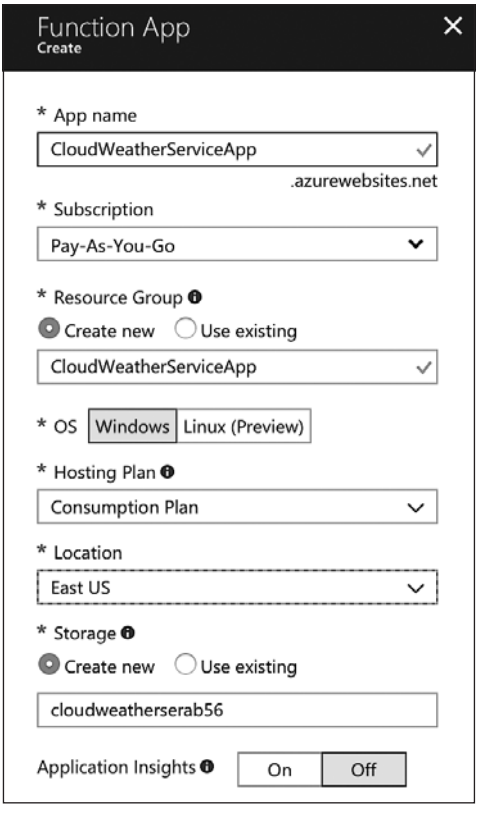


Рис. 10.11

Рис. 10.12

4. После создания функции откройте окно редактирования кода и используйте следующую логику на Python (рис. 10.13).

   Python-код для функции:

```
## импортируем нужные библиотеки
import os
import sys
import json
import argparse
import urllib
from urllib2 import HTTPError, URLError

## формируем и возвращаем ответ
def write_http_response(status, body_dict):
    return_dict = {
        "status": status,
        "body": body_dict,
        "headers": {
            "Content-Type": "application/json"
```

```
        }
    }
    output = open(os.environ['res'], 'w')
    output.write(json.dumps(return_dict))

## извлекаем значения входных параметров
zip = os.environ['req_query_zip']
countrycode = os.environ['req_query_countrycode']
apikey = os.environ['req_query_apikey']

print ("zip code value::" + zip + ", countrycode:" + countrycode + ",
apikey::" + apikey)

## формируем полный URL для обращения к сервису OpenWeatherMap
## с использованием подходящего ввода
baseUrl = 'http://api.openweathermap.org/data/2.5/weather'
completeUrl = baseUrl + '?zip=' + zip + ',' + countrycode + '&appid=' + apikey
print('Request URL--> ' + completeUrl)

## обращаемся к OpenWeatherMap API и разбираем ответ
## с надлежащей обработкой исключений

handling
try:
    apiresponse = urllib.urlopen(completeUrl)
except IOError as e:
    error = "IOError - The server couldn't fulfill the request."
    print(error)
    print("I/O error: {0}".format(e))
    errorcode = format(e[1])
    errorreason = format(e[2])
    write_http_response(errorcode, errorreason)
except HTTPError as e:
    error = "The server couldn't fulfill the request."
    print(error)
    print('Error code: ', e.code)
    write_http_response(e.code, error)
except URLError as e:
    error = "We failed to reach a server."
    print(error)
    print('Reason: ', e.reason)
    write_http_response(e.code, error)
else:
    headers = apiresponse.info()
    print('DATE :', headers['date'])
    print('HEADERS :')
    print('---------')
    print(headers)
    print('DATA :')
    print('---------')
    response = apiresponse.read().decode('utf-8')
    print(response)
    write_http_response(200, response)
```

Рис. 10.13

5. Сохраните функцию, добавьте простые параметры запросов (см. рис. 10.13) и проверьте свою функцию. Чтобы просмотреть все журналы и результаты выполнения функции, выберите опцию Monitor, расположенную под названием функции, — и увидите такую картину, как на рис. 10.14.
6. После модульного тестирования функции Azure на портале выберите опцию Integrate, находящуюся под названием функции, и убедитесь в том, что, помимо других настроек, включен метод HTTP GET, как показано на рис. 10.15.
7. Теперь пришло время создавать внешние API с использованием прокси-серверов. Для этого получите URL своей функции Azure, расположенный в верхнем правом углу окна редактирования функции (рис. 10.16).
8. Для того чтобы открыть эту функцию Azure через API, задайте настройки для прокси-сервера, нажав + напротив опции Proxies. Убедитесь, что там заданы настройки, отображенные на рис. 10.17. В поле Backend URL вставьте URL функции Azure, который вы получили в предыдущем шаге.
9. Бессерверный микросервис готов, и вы можете протестировать его в браузере или CLI, а это мы и рассмотрим уже в следующем разделе.

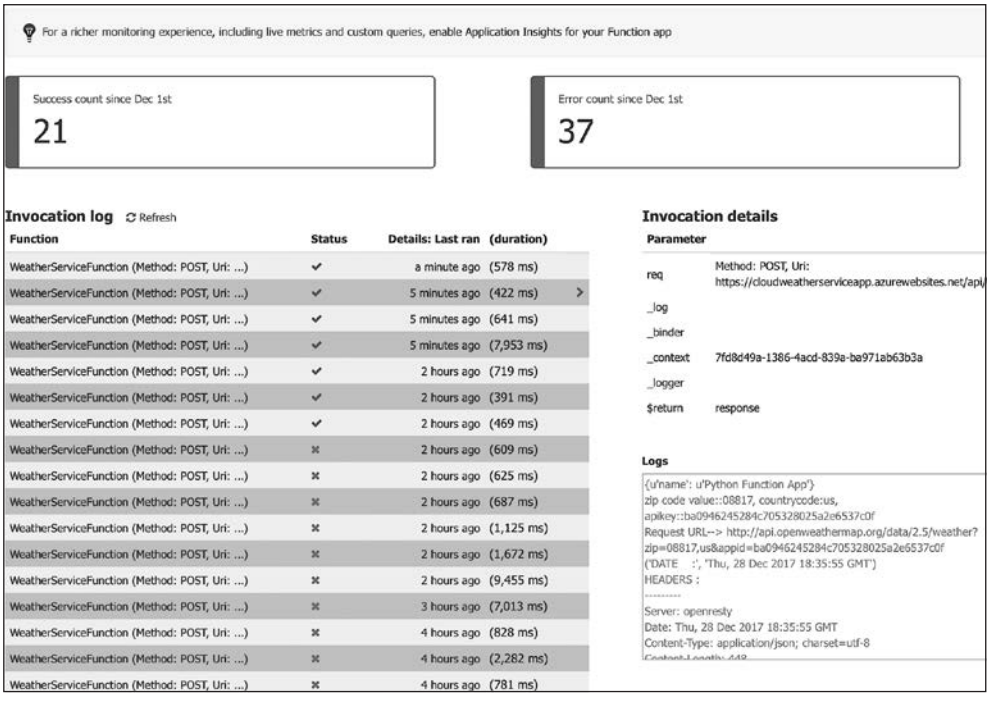


Рис. 10.14

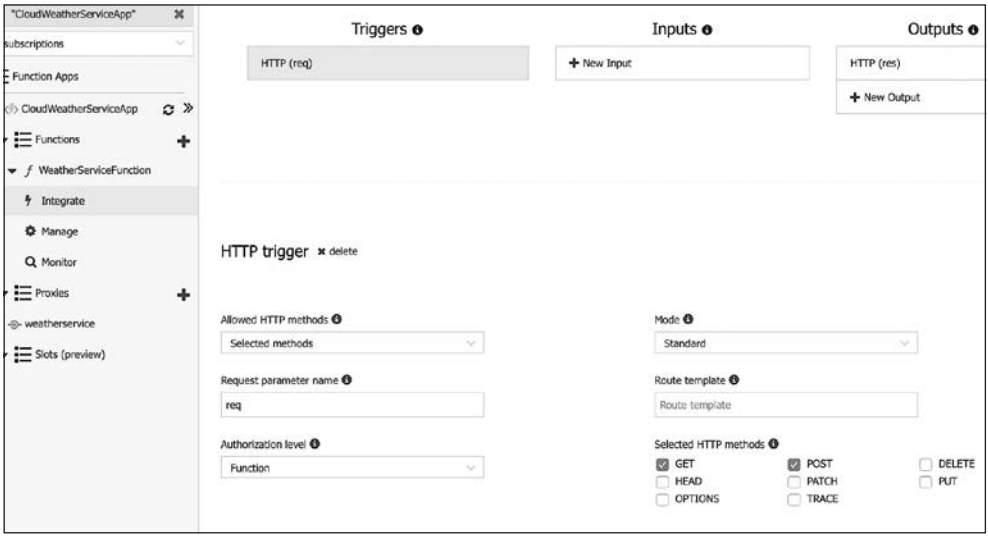


Рис. 10.15

Рис. 10.16

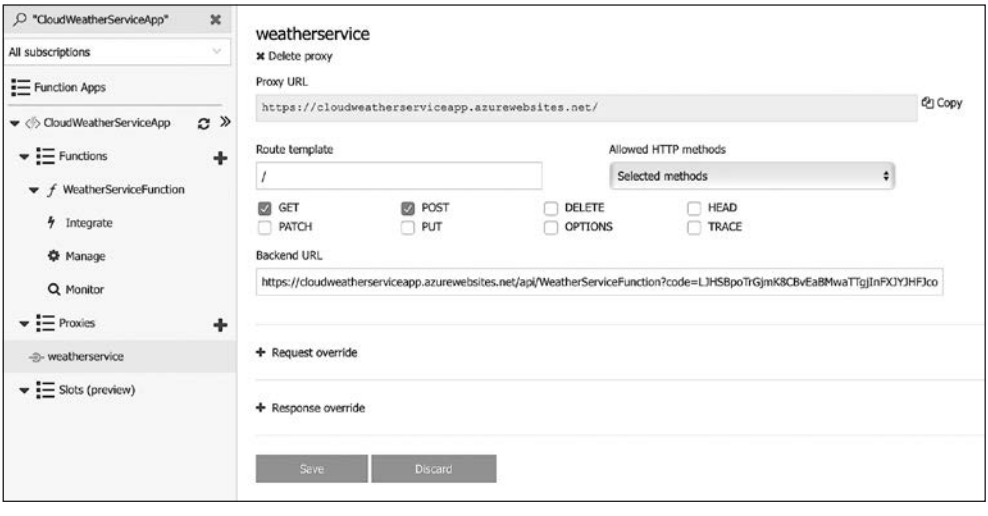


Рис. 10.17

Тестирование в браузере

Вы можете скопировать URL развернутого API из поля Proxy URL и попробовать по нему перейти, передавая параметры zip, countrycode и appid в составе URL:

```
https://asdfqwert.azurewebsites.net/?zip=10001&countrycode=us&apikey=
qwertyasdfg98765zxcv
```

Замените выделенный текст значениями, характерными для своего окружения.

Тестирование в CLI

Для этого вам нужно выполнить следующую команду:

```
$ curl -X GET
'https://asdfqwert.azurewebsites.net/?zip=10001&countrycode=us&apikey=
qwertyasdfg98765zxcv
```

Вы получите следующий результат:

```
"{"coord":{"lon":-74,"lat":40.75},"weather":[{"id":802,"main":"Clouds","descri
ption":"scattered clouds","icon":"03d"}],"base":"stations","main":{"temp":266.
73,"pressure":1024,"humidity":44,"temp_min":266.15,"temp_max":267.15},"visibil
ity":16093,"wind":{"speed":3.1,"deg":320},"clouds":{"all":40},"dt":1514582100,
"sys":{"type":1,"id":1969,"message":0.004,"country":"US","sunrise":1514549991,
"sunset":1514583454},"id":0,"name":"New York","cod":200}"
```

Автоматизация в Azure (ось 3 CNMM)

Концепция открытого облака основана на возможности использовать облачные услуги и интегрироваться с ними с помощью API, SDK и веб-сервисов на основе REST. Другой важный аспект этой парадигмы — наличие высокоуровневой автоматизации и управления сервисами, предоставляемых поставщиками облака, отчего становится еще легче оценить преимущества облачных вычислений, избегая излишнего вмешательства человека и позволяя создавать самовосстанавливающиеся и автоматически масштабируемые приложения. Кроме этого, разработчики теперь смогут скорее оперировать циклом обслуживания среды приложения, нежели полностью полагаться на команду администрирования, как было раньше. Это позволяет быстрее выпускать продукт на рынок и более гибко подходить к процессам цикла разработки приложения. Как говорилось в предыдущей главе, AWS предоставляет множество сервисов, которые можно использовать для реализации практик DevOps, автоматизации и других приемов разработки облачных приложений. Рассмотрим эти концепции в следующих разделах.

Инфраструктура как код

Как говорилось в предыдущей главе, одна из основных техник облачной автоматизации — это управление инфраструктурой как кодом для успешной реализации практик DevOps. Так же как в AWS есть Amazon CloudFormation, Azure предоставляет сервис *Azure Resource Manager (ARM)* Templates. С его помощью пользователи могут создавать стандартизированные и воспроизводимые JSON-шаблоны, которые применяются для обслуживания целых стеков приложения, выступающих целостной сущностью для ресурсной группы. Azure позволяет использовать интерфейс для непосредственного создания и редактирования ша-

блона, а если пользователи предпочитают разработку с помощью IDE, то в Visual Studio Code они найдут расширения, которые облегчают задачу создания ARM-шаблонов. Редактор в Azure помогает обнаружить в JSON синтаксические ошибки (пропущенные запятые, фигурные скобки и т. п.), но не семантические. Продолжая работать над своим приложением, вы можете развернуть готовый шаблон непосредственно на портале Azure либо с помощью CLI, PowerShell, SDK и т. д. Если в процессе развертывания шаблона возникают какие-либо ошибки, то с помощью ARM выполняются отладка и сбор данных, что очень полезно для оценки и исправления каких-либо дефектов. В отличие от AWS CloudFormation, шаблоны ARM не поддерживают формат YAML, более удобочитаемый и лаконичный по сравнению с JSON.

Для эффективного использования Visual Studio Code при разработке шаблонов ARM почитайте статью, располагающуюся по адресу https://docs.microsoft.com/en-us/archive/blogs/azuredev/iac-on-azure-developing-arm-template-using-vscode-efficiently.

Отличительной особенностью Azure Resource Manager выступает обслуживание подписки, когда владелец аккаунта может задать такие настройки, что вместе с подпиской должен будет указываться поставщик ресурсов. Некоторые поставщики ресурсов указываются по умолчанию, но в большинстве случаев пользователям необходимо обращаться к ним с помощью PowerShell-команд, таких как:

```
Register-AzureRmResourceProvider -ProviderNamespace Microsoft.Batch
```

У команды Azure есть репозиторий GitHub (github.com/Azure/azure-quickstart-templates/), где хранятся тысячи шаблонов для ARM. Поэтому, взяв оттуда базовый пример стека LAMP, можно с легкостью создать шаблон в ARM непосредственно на портале Azure (рис. 10.18).

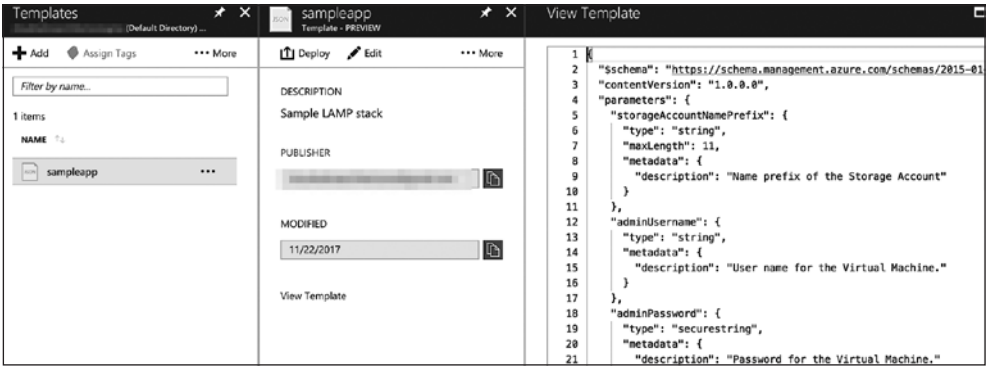


Рис. 10.18

Часть этого стека LAMP представлена далее (полный шаблон можно просмотреть на https://github.com/Azure/azure-quickstart-templates/tree/master/lamp-app):

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/
  deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "storageAccountNamePrefix": {
      "type": "string",
      "maxLength": 11,
      "metadata": {
        "description": "Name prefix of the Storage Account"
      }
    },
    "adminUsername": {
      "type": "string",
      "metadata": {
        "description": "User name for the Virtual Machine."
      }
    },
    ......
    ......
    ......
    ......

    {
      "type": "Microsoft.Compute/virtualMachines/extensions",
      "name": "[concat(variables('vmName'),'/newuserscript')]",
      "apiVersion": "2015-06-15",
      "location": "[resourceGroup().location]",
      "dependsOn": [
        "[concat('Microsoft.Compute/virtualMachines/', variables('vmName'))]"
      ],
      "properties": {
        "publisher": "Microsoft.Azure.Extensions",
        "type": "CustomScript",
        "typeHandlerVersion": "2.0",
        "autoUpgradeMinorVersion": true,
        "settings": {
          "fileUris": [
          "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/
          master/lamp-app/install_lamp.sh"
          ]
        },
        "protectedSettings": {
          "commandToExecute": "[concat('sh install_lamp.sh ',
          parameters('mySqlPassword'))]"
        }
      }
    }
  ]
}
```

В отличие от Amazon CloudFormation, ARM не предоставляет стандартной функции визуализации, но существуют инструменты, доступные в Сети, которые выполняют эту задачу. На рис. 10.19 представлена такая визуализация стека LAMP, составленная с помощью armviz.io/designer.

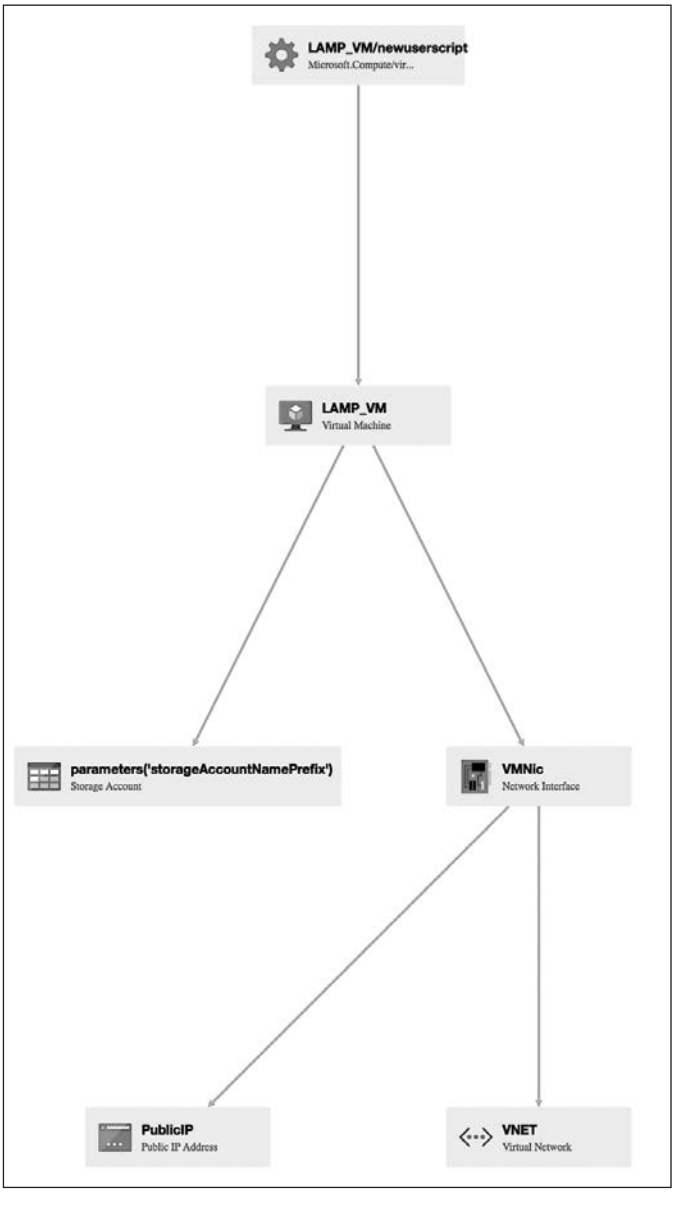


Рис. 10.19

Теперь же, как обсуждалось ранее, вы с легкостью можете задать целый стек в шаблоне ARM, но вам наверняка пригодится более продуманный пошаговый (последовательный или параллельный) подход к управлению разными системами и сервисами. Вам также может понадобиться возможность запускать этот процесс по возникновению некоторых событий, таких как внесение изменений на GitHub. И для такого рода управленческих задач Azure предлагает сервис под названием Azure Automation. Можете либо с его помощью создать текстовое представление перечня задач, где непосредственно приписываете PowerShell или Python-сценарий/модуль, задающий вашу бизнес-логику, либо, если не очень хотите иметь дело с кодом, сделать это через графическое представление перечня задач. В последнем случае у вас будет панель на портале Azure Automation, где можно создавать различные действия с широким рядом сущностей из библиотеки, задавать конфигурации для всех действий и связывать их вместе для запуска полного рабочего цикла. Перед непосредственным развертыванием этого процесса можете протестировать его самостоятельно и исправить недочеты или модифицировать, исходя из полученных результатов и данных журналов.

На рис. 10.20 показан пример графического представления перечня задач, доступного в Azure Automation Gallery, который соединяется с Azure посредством запуска автоматизированного процесса в качестве аккаунта и запускает все виртуальные машины V2, состоящие в подписке Azure, в ресурсной группе, или же одну виртуальную машину с названием V2 VM. Как видите, слева находятся командлеты PowerShell, другие перечни задач и ресурсы (реквизиты доступа, переменные,

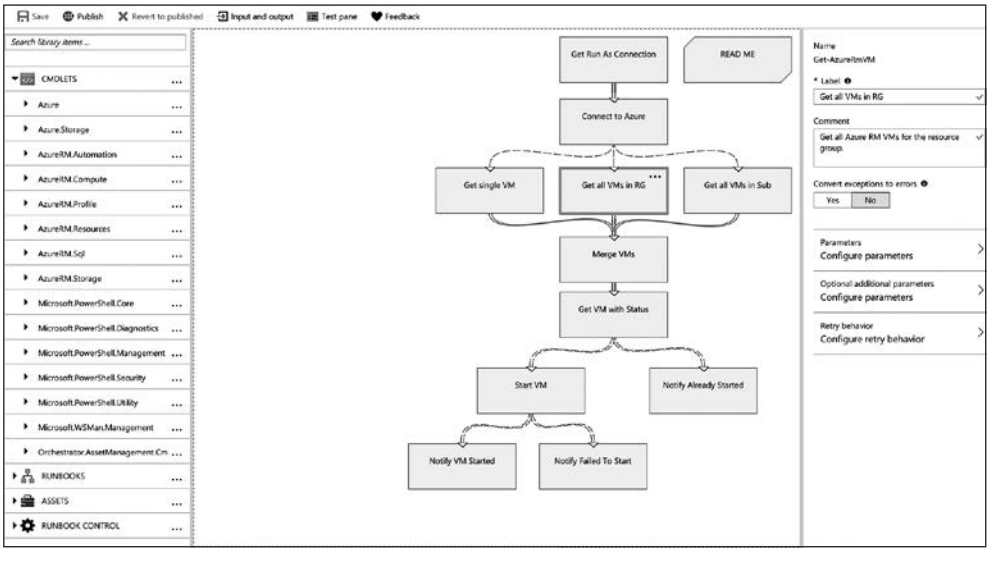


Рис. 10.20

соединения и сертификаты), которые можно использовать для макета. Справа — конфигурационная панель, где задаются и изменяются настройки и поведение для каждого действия в соответствии с требуемой бизнес-логикой.

CI/CD для бессерверных приложений

Как обсуждалось ранее в этой главе, вы можете легко создавать бессерверные приложения с помощью функций Azure. Разрабатывать и развертывать ваши Azure-функции можно непосредственно на портале Azure, однако это неэффективный подход с точки зрения командной работы и он не позволяет автоматизировать процессы согласно методологии DevOps. Поэтому Microsoft разработала ряд инструментов именно для VSTS-интеграции с функциями Azure, что облегчает управление всем CI/CD-конвейером. Есть и другие варианты с точки зрения источников развертывания, такие как BitBucket, Dropbox, OneDrive и т. д., но в этом разделе мы сосредоточимся на VSTS как одном из наиболее распространенных методов.

Перейдите по следующей ссылке, чтобы узнать подробнее об инструментах Visual Studio для функций Azure: https://devblogs.microsoft.com/aspnet/visual-studio-tools-for-azure-functions/.

Теперь, чтобы эффективно развернуть одну и ту же функцию в разных типах сред, крайне важно отделить основную логику от деталей конфигурации, характерных для конкретной среды. Чтобы сделать это, можете создать тип проекта с Azure-функциями в Visual Studio, в котором будет содержаться основная бизнес-логика в виде одной или нескольких функций. Чтобы управлять специфическими для окружения конфигурациями, их можно объединить в шаблоны ARM. Это поможет развертывать характерные для этапа Azure-функции в отдельных ресурсных группах, что обеспечивает четкое разделение, необходимое для реализации процессов CI/CD.

На рис. 10.21 представлена высокоуровневая схема архитектуры, разъясняющая данный процесс.

Как показано на схеме, в рамках непрерывной интеграции в большинстве проектов должны выполняться три основных шага (проекты можно изменять согласно вашим требованиям).

1. Сборка кода.
2. Выполнение юнит-тестирования, чтобы быть уверенными в правильности кода.
3. Создание пакета решения, который можно развернуть в соответствующих окружениях.

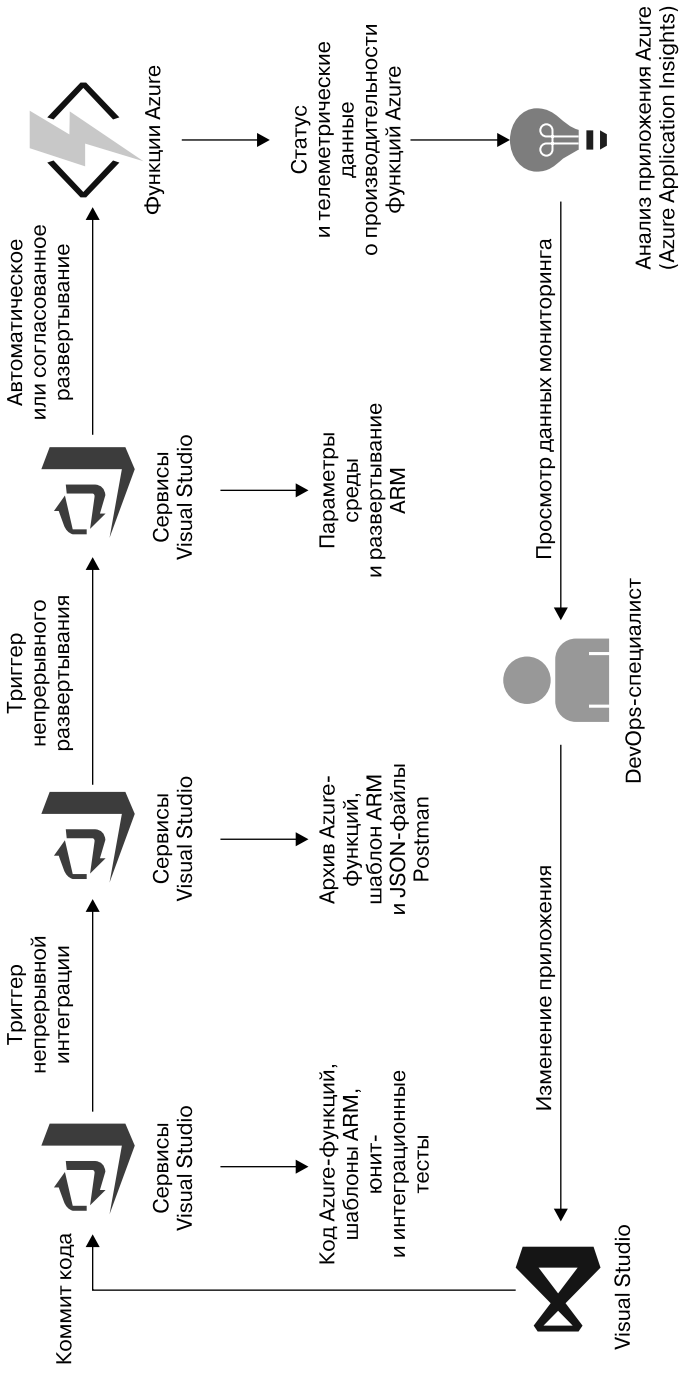

Коммит кода
Visual Studio
Триггер непрерывной интеграции
Сервисы Visual Studio
Код Azure-функций, шаблоны ARM, юнит- и интеграционные тесты
Изменение приложения
Триггер непрерывного развертывания
Сервисы Visual Studio
Архив Azure-функций, шаблон ARM и JSON-файлы Postman
DevOps-специалист
Автоматическое или согласованное развертывание
Сервисы Visual Studio
Параметры среды и развертывание ARM
Функции Azure
Статус и телеметрические данные о производительности функций Azure
Просмотр данных мониторинга
Анализ приложения Azure (Azure Application Insights)

Рис. 10.21

После завершения CI-процесса наступает время непрерывного развертывания. В большинстве случаев этот процесс состоит из нескольких фаз: Dev (deployment — развертывание), Test (testing — тестирование), UAT (User Acceptance Testing — приемочное пользовательское тестирование) и, наконец, выпуск готового продукта. Всеми ими можно управлять с помощью сервисов Visual Studio, где вы можете задать ход развертывания, включая параметры конфигурации среды и процесс подтверждения внесения изменений. Для низкоуровневых окружений, таких как Dev или Test, большинство клиентов применяют автоматическое развертывание, основанное на результатах юнит-тестов, но для высокоуровневых окружений, таких как UAT или промышленная эксплуатация, принято вручную подтверждать внесение изменений, так как это позволяет предохранять их от недочетов, обнаруженных по результатам тестирования низкоуровневых окружений. По завершении цикла DevOps результаты конечного развертывания изучаются с помощью Azure Application Insights, что в дальнейшем помогает исправлять и обновлять приложение, улучшая его функциональность в различных окружениях.

CI/CD для сервиса контейнеров Azure (Docker-контейнеры)

Microsoft предлагает управляемый сервис для Docker-контейнеров *Azure Container Service (AKS)*, с помощью которого клиенты смогут с легкостью развертывать расширяемые контейнерные приложения, работающие с открытыми API. Самая важная особенность AKS — то, что он обладает возможностью управлять контейнерами и размещать их. Это позволяет конечным пользователям сосредоточиться на своих приложениях, вместо того чтобы беспокоиться о запуске и масштабировании этих приложений вручную. Для управления предлагается несколько вариантов: Docker, Swarm, Kubernetes и Mesosphere DC/OS (рис. 10.22).

Другой важный аспект любой управляемой платформы для контейнеров — возможность пользоваться реестром контейнеров, в котором можно хранить все мастер-копии различных образов контейнеров согласно стекам приложения и конфигурации. Для этого Azure предлагает *Azure Container Registry (ACR)* — индивидуальный реестр, в котором вы можете помечать и загружать образы.

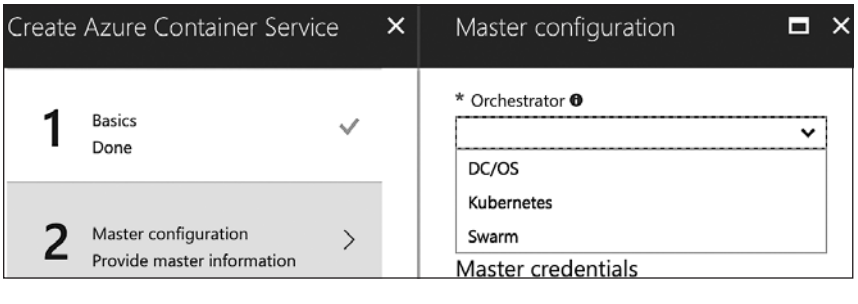


Рис. 10.22

Контейнерным приложениям зачастую нужно обращаться к внешним данным и хранить их во внешнем источнике. Azure-файлы можно использовать в качестве внешнего хранилища. См. docs.microsoft.com/ru-ru/azure/aks/azure-files-dynamic-pv.

Теперь, чтобы использовать практики CI/CD в Azure Container Service, мы выполним действия, очень похожие на те, что были описаны ранее в разделе о CI/CD для функций Azure. Далее представлены основные шаги и процесс внедрения этих практик.

1. Храните код своего приложения в репозитории, например в Git-репозитории сервисов Visual Studio. Кроме самого кода приложения, там же вы можете хранить файлы Docker и композиционные и разверточные файлы, которые понадобятся для создания образов Docker и развертывания согласно требованиям в различных окружениях.
2. Каждый раз, когда происходит коммит кода в репозиторий, запускается сборочный процесс непрерывной интеграции в VSTS и создаются образы Docker, которые в дальнейшем будут храниться в Azure Container Registry.
3. После внесения Docker-образов в реестр запускается новый релиз в VSTS, который последовательно можно передавать в Dev, Test, UAT и промышленную эксплуатацию.
4. Процесс релиза интегрирован с различными инструментами управления, поддерживаемыми AKS, которые получают последнюю версию образа Docker из реестра контейнеров и передают ее в запущенные инстансы, чтобы развернуть последнюю версию приложения.

В представленный процесс можно внести множество изменений, например внедрить возможность запуска разного рода тестов или интеграции с другими системами, что легко осуществить посредством CI/CD-конвейеров на VSTS.

Расширение Kubernetes для VSTS можно получить в Visual Studio Markerplace: marketplace.visualstudio.com/items?itemName=tsuyoshiushio.k8s-endpoint.

Методы перехода от монолитной архитектуры приложения к облачным архитектурам Azure

При миграции в облако разработчики зачастую придерживаются устоявшихся подходов и технологий, но каждый поставщик облака привносит в этот процесс свои особенности. Как упоминалось ранее, у AWS есть технология миграции 6Rs. Что касается Azure, то здесь технология довольно простая. В целом миграция разделена на три части (azure.microsoft.com/ru-ru/migration/).

- *Оценка*. В этой фазе вы оцениваете текущее состояние виртуальных машин, баз данных и приложений в среде эксплуатации и их миграционный путь с приоритетами.
- *Миграция*. Здесь происходит само перемещение ваших приложений, баз данных и VM в облако Azure с помощью нативных и сторонних инструментов, которые помогают вам перемещать данные посредством копирования.
- *Оптимизация*. После того как приложения и данные окажутся в облаке, вы сможете подстраиваться под необходимые расходы, производительность, безопасность и методы управления, применяя более продвинутые архитектуры и нативные облачные сервисы. Эта фаза в некоторой степени итеративная, так как оптимизацию необходимо выполнять регулярно по мере изменения приложения и появления новых облачных сервисов.

Для обслуживания представленных ранее фаз Azure предлагает ряд нативных облачных сервисов, а в качестве альтернативы — некоторые решения партнеров. Далее представлена сводная таблица этих возможностей. За наиболее свежей информацией обращайтесь сюда: azure.microsoft.com/ru-ru/migration/migration-partners/.

Фаза	**Нативные инструменты/сервисы Azure**	**Партнерские инструменты/сервисы**
Оценка	Azure Migrate (оценка VM), Data Migration Assistant (оценка баз данных)	Cloudamize, Movere, TSO Logic, CloudPhysics
Миграция	Azure Site Recovery (миграция VM), Azure Database Migration Service (миграция баз данных)	Cloudendure, Velostrata
Оптимизация	Cloudyn (теперь часть Microsoft)	Нет данных

Если говорить о гибридных облачных вычислениях, то помимо перечисленных выше инструментов Azure предлагает другой сервис обмена сообщениями по сравнению с AWS. AWS больше сосредоточена на нативных облачных сервисах и на том, как их можно использовать или расширять для управления ресурсами даже в частных локальных окружениях, тем самым искусственно создавая разрыв между частным и общедоступным облаком, где можно было бы формировать гибридные вычислительные окружения. Azure здесь оказалась на шаг впереди и предусмотрела возможность использования гибридного облака с помощью ключевого сервиса под названием Azure Stack. Вот так Azure описывает его: *«Azure Stack — это портфель продуктов, которые позволяют применять сервисы и возможности Azure в любой среде, от центров обработки данных до пограничных расположений и удаленных офисов. Создавайте и развертывайте гибридные и пограничные вычислительные приложения и согласованно запускайте их в пределах нужного расположения. Узнайте, как Azure Stack предоставляет гибкие возможности для выполнения различных рабочих нагрузок»* (см. azure.microsoft.com/ru-ru/overview/azure-stack/).

Ключевые области применения Azure Stack:

- граничные и отсоединенные от сети приложения;
- облачные приложения для обслуживания различных стандартов;
- модели облачных приложений, работающих локально.

Даже несмотря на то что эти области могут показаться захватывающими своей возможностью создания гибридной облачной среды с единым рядом инструментов управления и DevOps-процессов в рамках общедоступного облака Azure и Azure Stack, у них есть и обратная сторона медали, о которой пользователям стоит знать, если они планируют применять этот подход. Одним из основных испытаний здесь становится то, что Azure Stack возвращает пользователя обратно во времена обслуживания серверов и центров обработки данных, когда нужно было выполнять тяжелую работу по компоновке серверов для Azure Stack и обеспечивать их охлаждение, питание и бесперебойную работу. Поэтому, несмотря на то что Azure Stack позволяет развертывать контейнеры, микросервисы и PaaS-окружения локально, такой тип архитектуры сложно назвать по-настоящему облачным, так как вам все равно придется обслуживать всю инфраструктуру.

Резюме

В этой главе мы рассмотрели облачную платформу Microsoft Azure в рамках модели CNMM, о которой говорили ранее. Мы начали с основ Microsoft Azure, ее краткой истории, а затем погрузились в некоторые ее отличительные сервисы и предложения. После взглянули на аналогичное бессерверное микросервисное приложение, которое использовали в главе 9, и развернули его на Azure с помощью функций Azure и прокси-серверов. После этого мы сосредоточились на DevOps- и CI/CD-шаблонах и на том, как их связать с бессерверными приложениями и приложениями на Docker-контейнерах. И наконец, рассмотрели руководство по миграции на Azure: узнали, чем оно отличается от передачи сообщений AWS, а также каковы аспекты гибридного облака, связанного с Azure Stack.

В следующей главе попробуем разобраться в возможностях Google Cloud в сравнении с AWS и Azure.

11 Google Cloud Platform

Согласно множеству исследований аналитиков, третьим по значимости поставщиком облака является *Google Cloud Platform (GCP)*. История GCP началась в далеком 2008-м, когда компания Google запустила движок Google App, предназначенный для сообщества разработчиков, с упором на предложения типа *«платформа как услуга» (PaaS)*. Медленно и постепенно Google расширяла ряд предлагаемых сервисов и уже ближе к 2012 году начала уделять больше внимания темпам выпуска и расширению географии, что постепенно сделало ее одним из доминирующих игроков в этой сфере. С тех пор GCP расширила свое присутствие во множестве направлений, начиная от вычислений, хранения, обслуживания сетей и баз данных и заканчивая множеством высокоуровневых сервисов приложений, работающих в пространстве больших данных, IoT, *искусственного интеллекта*, API-платформ и различных экосистем.

Чтобы быть в курсе последних новостей GCP и узнавать о запусках новых сервисов, подпишитесь на блог GCP: cloud.google.com/blog/.

В этой главе мы рассмотрим следующее.

- Нативные сервисы Google Cloud Platform, их преимущества и отличительные черты в области CI/CD, особенности работы с бессерверными приложениями, контейнерами, концепциями микросервисов, включая следующие сервисы:
  - Cloud Kubernetes Engine;
  - Google Cloud Functions;
  - Cloud AI.
- Управление нативными архитектурами приложений GCP и мониторинг их возможностей.
- Методы для перехода от архитектур монолитных приложений к нативным архитектурам Google Cloud Platform.
- API, SDK, фреймворки с открытым исходным кодом и поддержка партнерских экосистем для создания нативных облачных приложений.
- Примеры эталонных архитектур и фрагментов кода для CI/CD, архитектур приложений на основе бессерверных микросервисов.

Облачные сервисы GCP (ось 1 CNMM)

Как было и в предыдущих главах, сначала попробуем разобраться в типах нативных облачных сервисов, предлагаемых Google Cloud, которые могут помочь конечным пользователям и бизнесу, так как предоставляют различные сервисы и платформы.

Введение

Компания Google вступила на рынок открытых облачных технологий немного позже других игроков, но за несколько лет нарастила потенциал с точки зрения как охвата услугами, так и их популярности у пользователей. Согласно различным аналитическим отчетам, по общей концепции и способности предоставлять заявленные возможности она занимает третье место среди облачных поставщиков (после AWS и MS Azure), что делает GCP многообещающим кандидатом для разработки и развертывания любых видов облачных приложений. Давайте взглянем на некоторые сервисы, предлагаемые пространством GCP, и на то, как их можно эффективно внедрять.

Google Cloud Platform: отличительные черты

У Google есть множество интересных сервисов в области *машинного обучения (ML)* и *искусственного интеллекта (AI)*, контейнеризации приложений и командной работы, о чем мы и поговорим в этом разделе. На самом деле многие из этих сервисов были изначально созданы в Google для внутренних нужд или даже обслуживания клиентской логики, но теперь из них сделали продукты и они стали доступны под эгидой Google Cloud. Давайте же погрузимся в некоторые ключевые концепции, популярные у пользователей.

Облачный AI

Google была одним из первых облачных поставщиков, который начал предлагать услуги в сфере искусственного интеллекта и машинного обучения, включая различные сервисы, предназначенные для широкого ряда пользователей, начиная с разработчиков и заканчивая специалистами по работе с данными. Большинство из этих сервисов/API были разработаны на основе внутренних инструментов и существующих продуктов, которые сейчас находятся в открытом доступе в рамках Google Cloud. Эти сервисы охватывают широкий ряд областей применения в AI/ML, например следующие.

Анализ изображений и видео. Одним из самых первых требований в организациях, начинающих использовать искусственный интеллект и машинное обучение, яв-

ляется возможность распознавания и классификации контекста и метаданных из изображений и потокового видео. В области анализа изображений Google предлагает Cloud Vision API (cloud.google.com/vision/), который поможет классифицировать изображения по тысячам категорий, обнаруживать определенные объекты и лица на изображениях, а также находить на них и читать напечатанные слова. Вы также можете выявлять нежелательный контент, находить логотипы, образы или даже похожие изображения. Аналогично для работы с видео Google предлагает Cloud Video Intelligence (cloud.google.com/video-intelligence/), который может найти любой момент в любом видеофайле в вашем каталоге, быстро добавить описание к видео, хранящимся в Google Cloud Storage, помогает определять ключевые объекты в видео и фиксировать моменты их появления. Благодаря этой видеоаналитике вы можете генерировать рекомендации по контенту для конечных пользователей и даже показывать больше контекстной рекламы, соответствующей содержанию видео.

AI-сервисы для работы с речью и текстом. Существует множество способов применения искусственного интеллекта, связанных с анализом текста и речи, включая преобразование текста в речь и наоборот. Именно поэтому Google разработала Cloud Speech API (cloud.google.com/speech-to-text), с помощью которого можно конвертировать аудио в текст с распознаванием более чем 110 языков и их вариантов, тем самым позволяя расшифровывать аудиоматериалы. Аналогичный продукт для преобразования письменного текста в речь известен под названием Cloud Text-To-Speech API (cloud.google.com/text-to-speech/). С его помощью можно синтезировать человекоподобную речь в различных языках и их вариантах, произносимую 30 голосами, доступными для воспроизведения. Кроме этого, Google предлагает сервис для обнаружения определенного языка в тексте и перевода этого текста на другой язык с помощью Cloud Translation API (cloud.google.com/translate/). Еще стоит упомянуть сервис Google, глубоко анализирующий текст: он извлекает информацию о людях, местах, событиях и о многом другом, встречающуюся в текстовых документах, статьях или сообщениях блогов. Этот сервис называется Cloud Natural Language (cloud.google.com/natural-language/), он помогает выявлять настроения и намерения (например, положительные или отрицательные отзывы) контента социальных сетей или сообщений клиентов в центре поддержки пользователей.

Чат-боты. Одна из самых распространенных областей применения искусственного интеллекта сегодня — создание диалоговых интерфейсов (или чат-ботов) для вебсайтов, мобильных приложений, мессенджеров и IoT-устройств, с возможностью естественного и насыщенного общения между пользователями и вашим бизнесом. Эти чат-боты должны понимать настроение и контекст беседы, что позволит им давать более понятные и точные ответы. Для этих целей компания в Google разработали сервис под названием DialogFlow Enterprise Edition (cloud.google.com/dialogflow), у него имеется ряд готовых шаблонов, он поддерживает более 20 языков, интегрируется с 14 различными платформами, что обогащает пользовательский опыт клиентов.

Кастомизация машинного обучения. В большинстве случаев опытные пользователи (например, специалисты по работе с данными) хотят расширить возможности управления своими алгоритмами, ML-моделями и тем, как система генерирует результаты. Рассмотренные ранее сервисы не обеспечивают нужного уровня контроля и настраиваемости, поэтому Google предлагает набор служб, таких как Cloud AutoML (cloud.google.com/automl/) и Cloud Machine Learning Engine (cloud.google.com/ai-platform), расширяющих эти возможности. Например, если продавец захочет классифицировать изображения различных платьев по цвету, форме, дизайну, он может воспользоваться Cloud AutoML: передать образцы данных для генерации пользовательских моделей машинного обучения, которые затем можно применять для создания алгоритмов распознавания реальных изображений. Аналогично, если специалистам по работе с данными нужно будет создать собственную модель для прогнозной аналитики, они могут использовать фреймворк, например TensorFlow, совместно с управляемым сервисом, таким как Cloud Machine Learning Engine, который уже интегрирован с множеством других сервисов Google и предоставляет привычный интерфейс в виде Jupiter notebooks, для создания пользовательских моделей. Задействуя эти сервисы вместе с Cloud TPU (модуль обработки для TensorFlow, cloud.google.com/tpu), можно достичь производительности до 180 терафлопс, предоставляя вычислительные мощности для обучения и развертывания современных моделей машинного обучения в больших масштабах.

Чтобы оперативно узнавать о последних новостях Google Cloud в сфере больших данных и машинного обучения, подпишитесь на следующий блог: cloud.google.com/blog/products/data-analytics.

Kubernetes Engine

В течение последних нескольких лет сообщество разработчиков приобщилось к практике использования контейнеров (например, Docker) для развертывания приложений в форме легковесных и самодостаточных образов, которые включают в себя все необходимое для их запуска: код, среду выполнения, системные инструменты, системные библиотеки и настройки. Сегодня для обеспечения эффективной масштабной работы контейнеров необходимо иметь платформу для управления, которая поможет обслуживать такие функции, как масштабирование кластеров в соответствии с нагрузкой, самовосстановление поврежденных модулей, настройки и управление лимитами ресурсов (например, CPU и RAM), что позволит внедрить интегрированный мониторинг и возможности журналирования — как раз то, для чего предназначен Kubernetes (известный как K8s). Google создала Kubernetes и много лет использовала его для внутренних нужд для развертывания контейнеров, прежде чем он оказался открытым и стал частью Cloud Native Computing Foundation.

Если вы фанат комиксов, то вам стоит взглянуть на комикс о Kubernetes Engine: cloud.google.com/kubernetes-engine/kubernetes-comic/.

Kubernetes основан на тех же принципах проектирования, что и другие популярные сервисы Google, и у него те же преимущества: автоматизированное управление, мониторинг и сбор информации об устойчивости контейнеров приложения, автоматическое масштабирование, постепенные обновления и т. д.

Вы можете быстро ознакомиться с консолью Google Cloud или API/CLI и развернуть свой первый кластер, как показано на рис. 11.1. Можете также воспользоваться Google Cloud Shell для развертывания тестового приложения, чтобы ознакомиться с особенностями работы с кластерами.

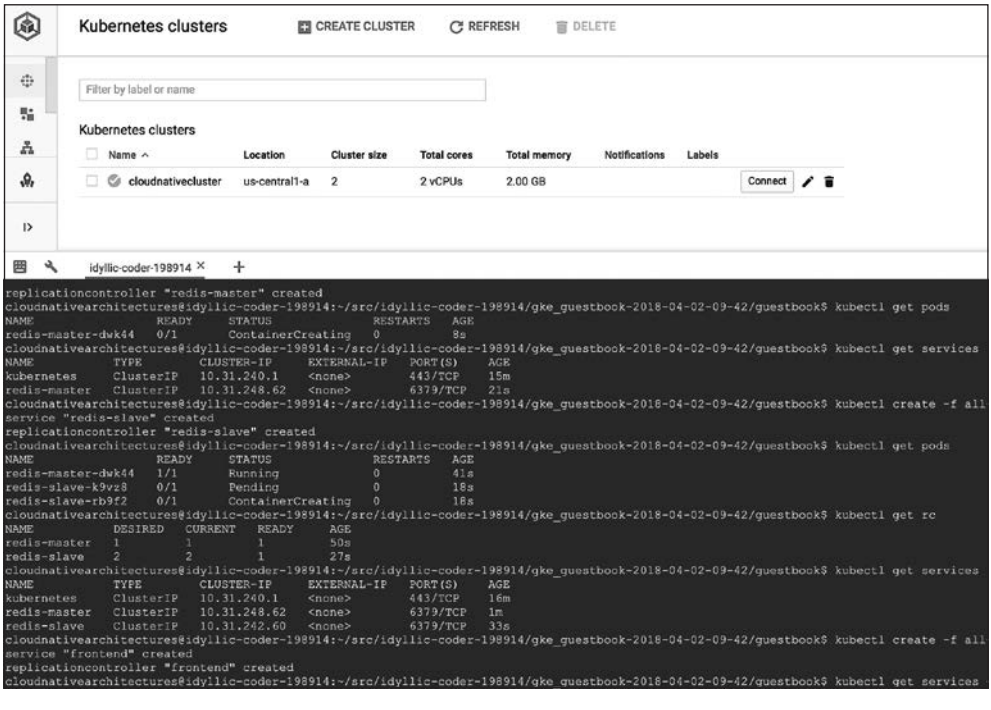


Рис. 11.1

Google наряду с Kubernetes Engine поставляет также сервис Cloud Build (cloud.google.com/cloud-build), с помощью которого вы можете упаковать свои приложения и другие артефакты в Docker-контейнер для последующего развертывания. Поскольку сервис поддерживает автоматические триггеры на изменения в репозиториях исходного кода, таких как GitHub, Bitbucket и репозитории Google Cloud Source, то вы сможете создавать полностью автоматизированные CI/CD-конвейеры, которые

можно запускать при каждом внесении изменений в репозиториях. Еще один сервис, связанный с экосистемой контейнеров, — Container Registry (cloud.google.com/container-registry/), он позволяет хранить частные образы контейнеров, загружать, скачивать их, а также управлять ими из любой системы, инстанса виртуальной машины или любого вашего оборудования.

G Suite

Google уже давно поставляет пользователям онлайн-инструменты для совместной работы, увеличивающие продуктивность. Но чтобы оставить след в мире корпоративных инструментов, в Google разработали G Suite (сейчас известен как Google Workspace) (workspace.google.com), предлагают его наряду с другими облачными инструментами и с тех пор получают благодарность множества пользователей (рис. 11.2, источник — workspace.google.com/features/).

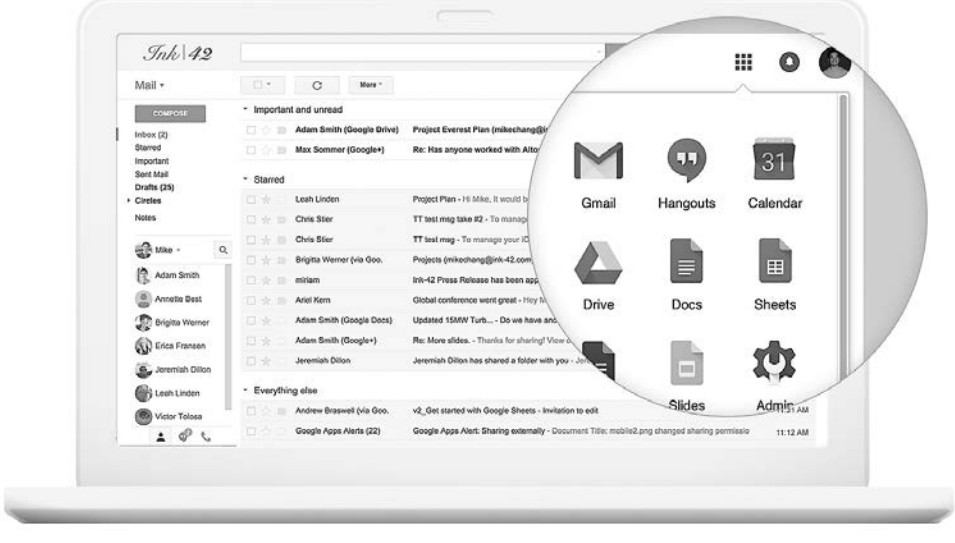


Рис. 11.2

В целом G Suite (Google Workspace) предлагает сервисы из четырех категорий.

- *Интеграция*. Существует ряд сервисов, таких как Gmail, календари, Google+ и Hangouts, которые позволяют улучшить связи внутри организации.
- *Создание*. Для улучшения взаимодействия Google поставляет такие сервисы, как Docs, Sheets, Forms, Slides, Sites и Jamboard, которые можно применять для создания и распространения контента.

- *Доступ*. Часто пользователям необходимо создавать резервные копии и распространять файлы, видео и другие медиафайлы с применением безопасного облачного хранилища, где как раз и пригодятся Google Drive и Cloud Search.
- *Управление*. Эти сервисы относятся к категории управленческих и позволяют на корпоративном уровне осуществлять централизованный контроль, например, за управлением пользователями, политикой безопасности, резервным копированием и хранением, а также управлением мобильными устройствами.

У G Suite (Google Workspace) есть официальный блог, в котором сообщаются новости об обновлениях и новой функциональности: workspaceupdates.googleblog.com/.

Проектирование, ориентированное на приложения (ось 2 CNMM)

Изучив ряд интересных сервисов Google Cloud, давайте погрузимся непосредственно в создание облачных архитектур приложений, применяя для этого лучшие практики.

Бессерверные микросервисы

Подобно тому как мы поступали в предыдущих главах об AWS и MS Azure, рассмотрим создание бессерверного микросервиса с помощью сервисов Google Cloud. Но прежде стоит упомянуть о портфолио бессерверных сервисов Google, представленных на рис. 11.3 (источник — cloud.google.com/serverless/whitepaper/), где показаны ранние сервисы, такие как App Engine, и самые последние, включая Cloud Functions и Cloud Machine Learninf Centre. Чтобы узнать об этом подробнее, обращайтесь к технической документации и информации на портале Google Cloud.

Что касается самого приложения, мы снова обратимся к примеру создания бессерверного приложения прогноза погоды, которое неявно обращается к OpenWeatherMap API. Как и в предыдущих главах, задействуем возможности облачных функций Google Cloud, однако пока не сможем применить Google Cloud Endpoints (управляемый API-сервис), так как на время написания книги еще не была внедрена интеграция этих сервисов. Функции Google Cloud сейчас поддерживают только JavaScript и выполняются в среде Node.js, поэтому перенесем написанную ранее бизнес-логику на JavaScript, чтобы иметь возможность продемонстрировать эту область применения. На рис. 11.4 представлен пример архитектуры, использованный для создания приложения.

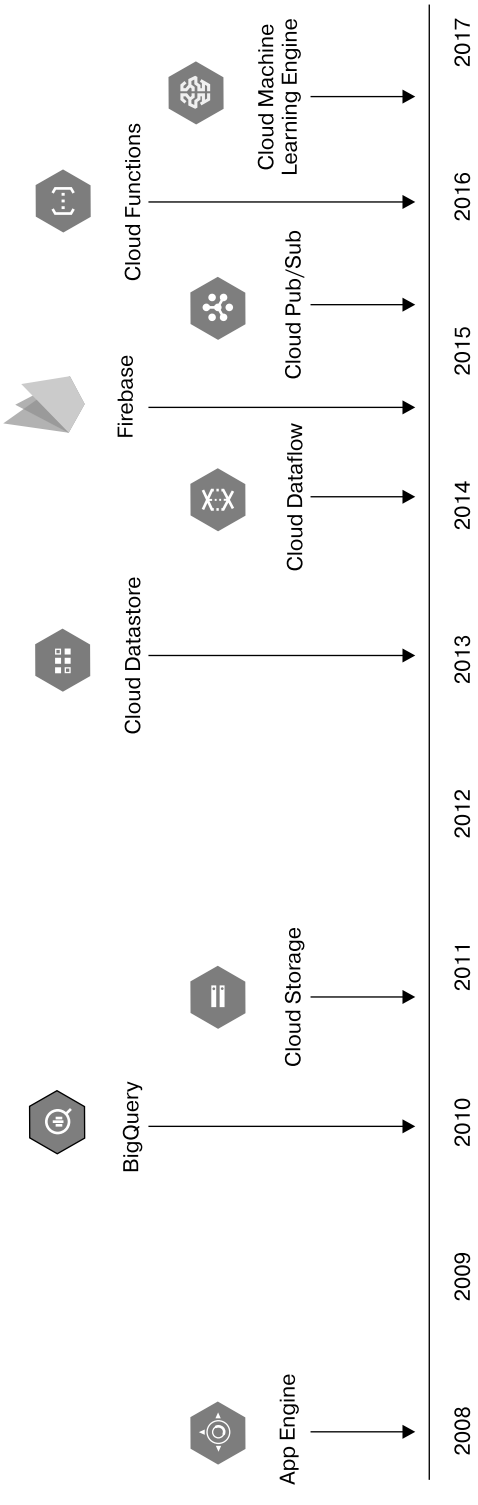

2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
App Engine
BigQuery
Cloud Storage
Cloud Datastore
Cloud Dataflow
Firebase
Cloud Pub/Sub
Cloud Functions
Cloud Machine Learning Engine

Рис. 11.3

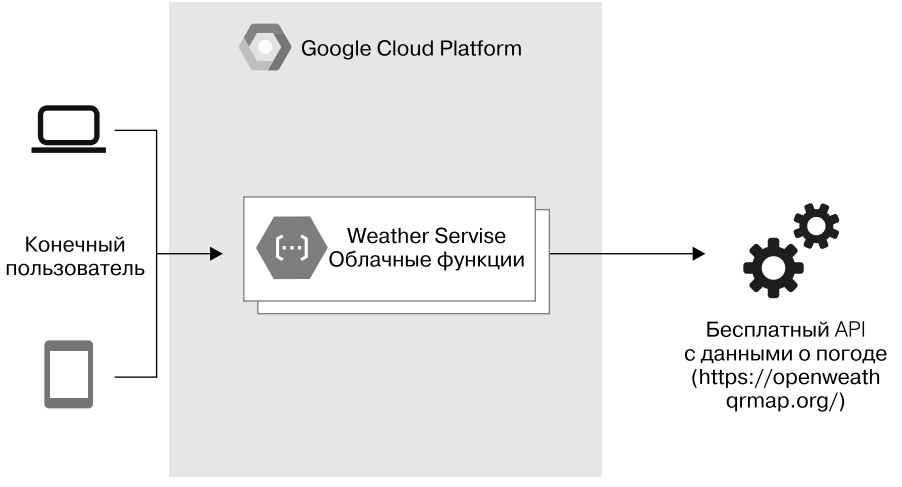


Рис. 11.4

Пример

Как и в предыдущих главах, будем создавать бессерверный микросервис с помощью функций Google Cloud, поэтому придерживайтесь следующих шагов.

1. Перейдите в консоль Google Cloud и найдите категорию Cloud Functions (рис. 11.5).

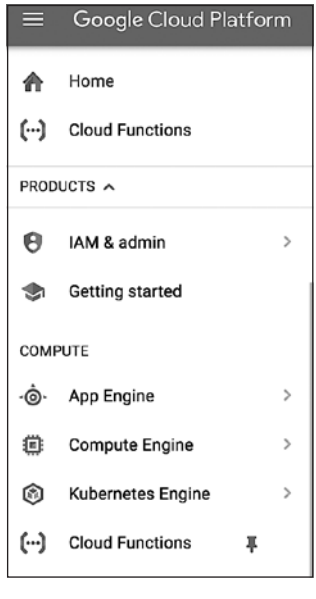


Рис. 11.5

2. Выберите вариант Create Functions и начните задавать функцию HTTP-триггера (рис. 11.6).

Рис. 11.6

3. Напишите в области `index.js` следующий код в качестве основной бизнес-логики для нашего бессерверного микросервиса (заметим, что прямо под кодом в консоли необходимо установить для Function to execute значение `weatherService`):

```
function handlePUT(req, res) {
 // Возвращаем сообщение Forbidden для POST-запроса
 res.status(403).send('Forbidden!');
 }

function handleGET(req, res) {
 // Извлекаем параметры URL-адреса
 var zip = req.query.zip;
 var countrycode = req.query.countrycode;
 var apikey = req.query.apikey;

 // Создаем полный URL для OpenWeatherMap с помощью параметров,
 // предоставленных пользователем
 var baseUrl = 'http://api.openweathermap.org/data/2.5/weather';
 var completeUrl = baseUrl + "?zip=" + zip + "," + countrycode + "&appid="
 + apikey;
 console.log("Request URL--> " + completeUrl)

 // Импортируем модуль sync-request для отправки HTTP-запросов
 var weatherServiceRequest = require('sync-request');

 // Обращаемся к OpenWeatherMap API
 var weatherServiceResponse = weatherServiceRequest('GET', completeUrl);
 var statusCode = weatherServiceResponse.statusCode;
 console.log("RESPONSE STATUS -->" + statusCode);

 // Проверяем, был ответ положительным или мы получили ошибку
 if (statusCode < 300) {
 console.log("JSON BODY DATA --->>" + weatherServiceResponse.getBody());
 // Если ответ был успешным, возвращаем код состояния, тип и тело содержимого
 console.log("Setting response content type to json");
 res.setHeader('Content-Type', 'application/json');
 res.status(statusCode);
 res.send(weatherServiceResponse.getBody());
 } else {
 console.log("ERROR RESPONSE -->" + statusCode);
 // Если мы получили ошибку, отправляем обратно информацию о ней
 res.status(statusCode);
 res.send(statusCode);
 }
 }

/**
 * Возвращаем информацию о погоде в ответ на GET-запрос.
   Запрещаем PUT-запросы.
 *
 * @param {Object} req Контекст запроса облачной функции.
```

```
* @param {Object} res Контекст ответа облачной функции.
*/
exports.weatherService = (req, res) => {
switch (req.method) {
case 'GET':
handleGET(req, res);
break;
case 'PUT':
handlePUT(req, res);
break;
default:
res.status(500).send({
error: 'Something blew up!'
});
break;
}
};
```

4. Теперь измените параметр `package.json`, включив в него соответствующие зависимости (рис. 11.7).

Рис. 11.7

Далее представлен код, который нужно разместить в окне `package.json`:

```
{
 "name": "sample-http",
 "version": "0.0.1",
 "dependencies": {
 "sync-request": "^2.0"
 }
}
```

5. Нажмите кнопку Create после того, как проверите все настройки и код на соответствие предыдущим шагам. Через несколько минут функция будет создана и станет доступна в консоли.
6. Теперь нужно проверить функцию. Для этого перейдите на страничку подробностей и под параметром Trigger найдете конечную точку HTTPS для вашей функции, которую можно использовать для вызова и тестирования микросервиса. Скопируйте этот URL, он пригодится в следующем шаге (рис. 11.8).

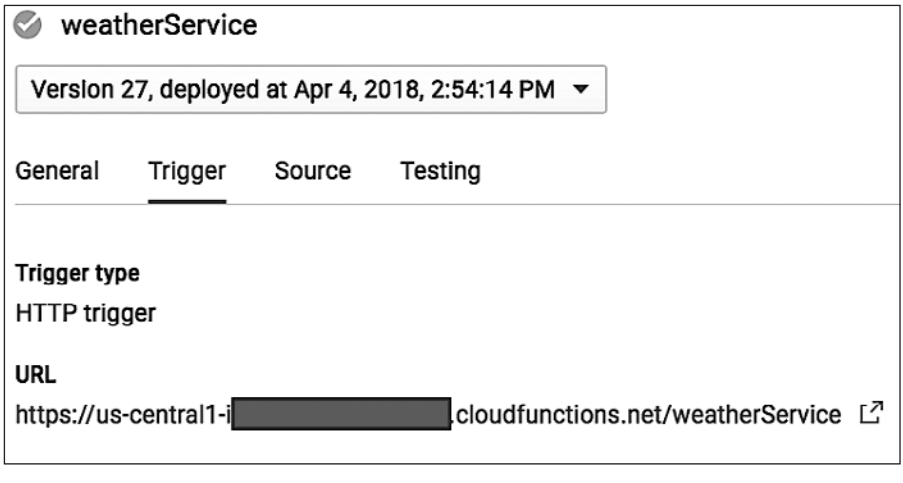


Рис. 11.8

7. Добавьте в ранее скопированный URL следующие параметры: `zip`, `countrycode` и `apikey`, как в примере:

```
us-central1-abcdef-43256.cloudfunctions.net/weatherService/?zip=10011&count
rycode=us&apikey=vjhvjvjhvjhv765675652hhjvjsdjysfydFjy
```

Теперь откройте любой браузер, перейдите по этой ссылке, и вы увидите магию своими глазами! Функция успешно выполняется, и после этого вы можете просмотреть JSON-ответ с информацией о погоде для местности, которую указали в параметрах.

8. Можете проверить то же самое с помощью команды `curl`:

```
$ curl -X GET
'us-central1-idyllic-coder-198914.cloudfunctions.net/weatherService/?zip=
10001&countrycode=us&apikey=098172635437y363535'{"coord":{"lon":-73.99,
"lat":40.73},"weather":[{"id":500,"main":"Rain","description":"light rain",
"icon":"10d"}],"base":"stations","main":{"temp":285.07,"pressure":998,
"humidity":87,"temp_min":284.15,"temp_max":286.15},"visibility":16093,
"wind":{"speed":10.8,"deg":290,"gust":14.9},"clouds":{"all":90},"dt":
1522869300,"sys":{"type":1,"id":1969,"message":0.0059,"country":"US",
"sunrise":1522837997,"sunset":1522884283},"id":420027013,"name":
"New York","cod":200}
```

9. После тестирования функции вы сможете мониторить результаты из консоли, для чего перейдите на вкладку General в деталях функции. Там будут доступны множество видов графиков, например составленных на основе *частоты вызовов* (рис. 11.9), *использовании памяти*, *времени исполнения* (рис. 11.10).

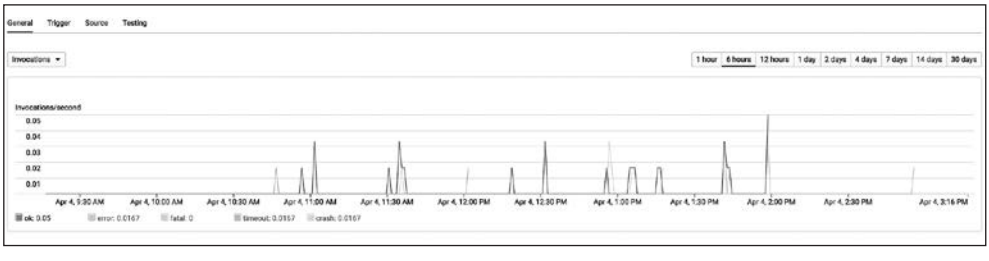


Рис. 11.9

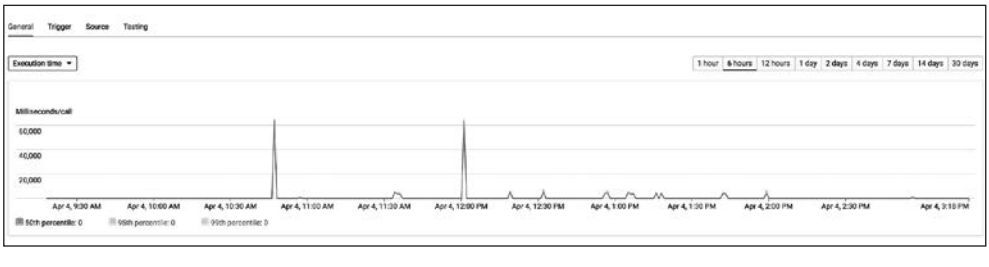


Рис. 11.10

10. Еще один полезный инструмент для управления вашими функциями Google — *журналы* Stackdriver, которые могут предоставить информацию для отладки на бэкенде по коду и исполнению функции (рис. 11.11).

Здесь завершим ознакомление с примером проектирования тестового бессерверного микросервиса с помощью функций Google.

Рис. 11.11

Автоматизация в Google Cloud Platform (ось 3 CNMM)

Как говорилось в предыдущих главах, автоматизация — это ключевое требование для организации оптимальных процессов развертывания ваших приложений в облаке и управления ими. Для обеспечения этой возможности Google Cloud, помимо многих сервисов, помогающих автоматизировать рабочие процессы, стал предлагать широкий ряд API и SDK. Фактически, исходя из индивидуальных предпочтений разработчиков, можно выбрать из нескольких вариантов для клиентских библиотек SDK, включая Java, Python, NodeJS, Ruby, Go, .Net и PHP, которые обеспечивают гибкость и простоту автоматизации и интеграции. Поэтому в следующем разделе мы дополнительно рассмотрим этот аспект и обратим внимание на различные варианты автоматизации и реализации практик DevOps.

Инфраструктура как код

Google Cloud предлагает очень похожий на CloudFormation от AWS сервис под названием Google Cloud Deployment Manager. С его помощью можно легко в декларативном виде (YAML) описывать инфраструктурные компоненты для автоматизации различных задач, связанных с обслуживанием и автоматизацией. Поэтому, используя этот сервис, вы сможете создавать шаблоны, которые помогут приложению обслуживать и развертывать множество компонентов. Например, если у вас трехуровневое приложение, которое самостоятельно масштабируется и оснащено распределителями нагрузки, а также распоряжается данными из баз данных, вы могли бы написать YAML-шаблон для Cloud Deployment Manager, что поможет автоматизировать весь процесс. Чтобы сделать его более гибким, вы можете задать возможность задействовать параметры шаблона и переменные среды, которые можно будет указывать при запуске. Масштабы окружений Test/Dev

могут быть небольшими, но в промышленной среде они должны быть полноценно масштабируемыми.

Есть у модулей шаблонов и составных типов еще пара интересных функций, которые могут помочь значительно расширить функциональность Deployment Manager. В качестве составляющей модулей шаблонов вы можете написать файлы-помощники на Python или на Jinja, чтобы выполнять особые функции, например генерировать уникальные имена ресурсов, и тем самым улучшить шаблоны. С помощью механизма использования Jinja- или Python-кода вы сможете создавать составные типы, которые позволяют иметь один или много шаблонов, настроенных на совместную работу. Так, например, вы можете разработать составной шаблон — в сущности, специфическую конфигурацию сети VPC, который можно задействовать каждый раз, когда вы создаете новую среду выполнения приложений.

Перейдите по следующей ссылке, чтобы узнать последние новости о поддерживаемых ресурсах Cloud Deployment Manager: cloud.google.com/deployment-manager/docs/configuration/supported-resource-types.

Теперь взглянем на пример приложения и на то, как это все можно использовать для создания ресурсов в Google Cloud. Для того чтобы было проще начинать, можете просмотреть примеры кода, которые Google размещает на GitHub. Возьмем оттуда пример и сразу перейдем к демонстрации самой концепции. Пример кода, который создает виртуальную машину в специальном Google Cloud-проекте, вы найдете по следующей ссылке: github.com/GoogleCloudPlatform/deploymentmanager-samples/blob/master/examples/v2/quick_start/vm.yaml.

Как сказано в примечаниях к этому шаблону, не забудьте указать значения, характерные для вашего проекта, например ID проекта вместо `MY_PROJECT` и имя семьи инстансов вместо `FAMILY_NAME`, чтобы обеспечить должное обслуживание инстанса с помощью менеджера развертывания.

Один из самых быстрых способов — сделать это с помощью Google Cloud Shell, которая поставляется вместе с предустановленным консольным интерфейсом gcloud и не требует серьезной настройки. Как показано на рис. 11.12, мы используем шаблон для предоставления ресурсов, и команда выполняется успешно.

```
cloudnativearchitectures@idyllic-coder-198914:~$ gcloud deployment-manager deployments create simple-deployment --config vm.yaml
The fingerprint of the deployment is 7y7pOqjcF6g7CpIOsqlphQ==
Waiting for create [operation-1525443310211-56b61f2a14fb8-7a136a28-eaa7aa5b]...done.
Create operation operation-1525443310211-56b61f2a14fb8-7a136a28-eaa7aa5b completed successfully.
NAME                        TYPE                   STATE      ERRORS  INTENT
quickstart-deployment-vm    compute.v1.instance    COMPLETED  []
```

Рис. 11.12

Создав ресурсы, можете использовать консоль gcloud для их описания (рис. 11.13).

Помимо консоли glcoud, можете воспользоваться Google Cloud Console, а затем перейти в раздел Cloud Deployment Manager к таким же результатам развертыва-

ния, для того чтобы получать подробности, щелкая на разных разделах и ссылках (рис. 11.14).

```
cloudnativearchitectures@idyllic-coder-198914:~$ gcloud deployment-manager deployments describe simple-deployment
---
fingerprint: 7y7p0qjcF6g7Cp1OsqlphQ==
id: '6126455663787889665'
insertTime: '2018-05-04T07:15:10.302-07:00'
manifest: manifest-1525443310334
name: simple-deployment
operation:
  endTime: '2018-05-04T07:15:46.580-07:00'
  name: operation-1525443310211-56b61f2a14fb8-7a136a28-eaa7aa5b
  operationType: insert
  progress: 100
  startTime: '2018-05-04T07:15:10.880-07:00'
  status: DONE
  user: cloudnativearchitectures@gmail.com
NAME                    TYPE                  STATE      INTENT
quickstart-deployment-vm  compute.v1.instance  COMPLETED
```

Рис. 11.13

Рис. 11.14

CI/CD для бессерверных микросервисов

Google Cloud все еще находится на ранних стадиях развития, если говорить об инструментах для работы с CI/CD для бессерверных микросервисов. И хоть для управления исходным кодом Google предлагает сервис Cloud Source Repositories (cloud.google.com/source-repositories), который можно использовать и для управления кодом Cloud-функций, на момент написания этой книги возможности Google по созданию полноценных CI/CD-окружений ограничены.

Один из вариантов, помогающих обслуживать этот процесс, — бессерверный фреймворк, который имеет встроенный плагин для работы с функциями Google Cloud и помогает легко создавать, устанавливать, упаковывать и развертывать функции, связанные с бизнес-логикой. Подробнее об этом можно прочитать здесь: github.com/serverless/serverless-google-cloudfunctions.

Также есть пример, который поможет вам быстрее начать работу: www.serverless.com/framework/docs/providers/google/examples/.

CI/CD для контейнерных приложений

Один из самых популярных способов развертывания контейнерных приложений в Google Cloud — это использование Kubernetes. Как говорилось ранее, Kubernetes — это продукт внутренних технологий Google, разработанный как раз для решения таких проблем, как непрерывная интеграция кода и развертывание. Поэтому поиски методов реализации CI/CD-шаблонов в таком окружении обычно заканчиваются на нем.

Реализация механизмов управления выпусками не должна составить труда, если применять Kubernetes CLI/API, так как у него есть функции, которые помогают обновлять ресурсы согласно последней версии кода приложения с использованием последних образов контейнеров. Сделать это можно и с помощью механизма внесения изменений, а если в ходе обновления возникли ошибки, то можно прервать процесс и откатиться до предыдущей версии. Подробнее об операциях обновления у Kubernetes CLI читайте здесь: kubernetes.io/docs/reference/kubectl/cheatsheet/#updating-resources.

Есть интересный комикс о Google Cloud, который помогает понять возможности Kubernetes для CI/CD: cloud.google.com/kubernetes-engine/kubernetes-comic/.

Еще одним механизмом реализации CI/CD в среде Kubernetes является использование Jenkins — сервера автоматизации с открытым исходным кодом, который позволяет гибко управлять вашими конвейерами сборки, тестирования и развертывания. Чтобы его использовать, нужно развернуть Jenkins на Kubernetes Engine (версия Kubernetes для Google Cloud), установить плагины для Jenkins, задать нужные конфигурации для управления процессом развертывания. На рис. 11.15 показаны процесс установки и отдельные шаги согласно документации Google (источник — cloud.google.com/solutions/continuous-delivery-jenkins-kubernetes-engine).

Методы перехода от монолитных архитектур приложений к архитектурам Google Cloud

В предыдущих разделах мы рассматривали приложения, создаваемые с нуля, и говорили о том, как использовать нативные облачные возможности, такие как бессерверные вычисления, контейнеры, микросервисные архитектуры, шаблоны CI/CD и т. п. В типичных же корпоративных окружениях заказчики обычно уже вложили серьезные инвестиции в среды локального или совместного размещения оборудования, поэтому для получения выгоды нужно будет переместить вычислительные мощности в облако. Google Cloud предлагает нативные сервисы, а также партнерские решения, которые можно применять на разных стадиях миграции. В целом в Google говорят, что перенос любого проекта в облако складывается из четырех фаз: *оценки*, *планирования*, *настройки сетей* и *копирования*.

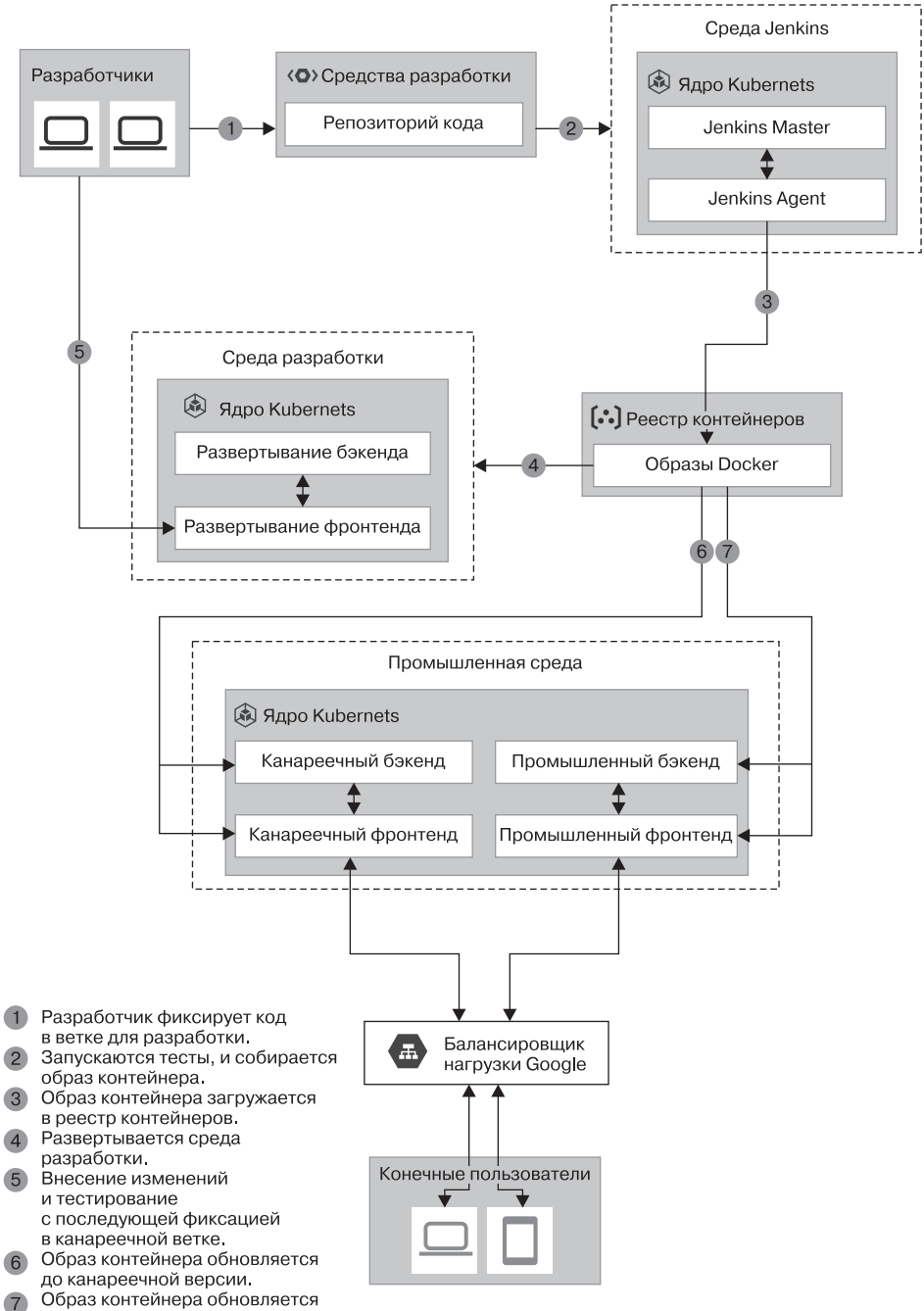

Разработчики
Средства разработки
Репозиторий кода
Среда Jenkins
Ядро Kubernetes
Jenkins Master
Jenkins Agent
Среда разработки
Ядро Kubernetes
Развертывание бэкенда
Развертывание фронтенда
Реестр контейнеров
Образы Docker
Промышленная среда
Ядро Kubernetes
Канареечный бэкенд
Промышленный бэкенд
Канареечный фронтенд
Промышленный фронтенд
Балансировщик нагрузки Google
Конечные пользователи
1 Разработчик фиксирует код в ветке для разработки.
2 Запускаются тесты, и собирается образ контейнера.
3 Образ контейнера загружается в реестр контейнеров.
4 Развертывается среда разработки.
5 Внесение изменений и тестирование с последующей фиксацией в канареечной ветке.
6 Образ контейнера обновляется до канареечной версии.
7 Образ контейнера обновляется до промышленной версии

Рис. 11.15

В большинстве случаев во время оценки и настройки сетей заказчики сосредоточены на поиске подходящих инструментов, разработке процессов автоматизации — всего того, что помогает выявлять существующую рабочую нагрузку в локальной среде, а также выборе подходящих настроек сети, чтобы обойти острые углы в ходе миграции.

Для этапа планирования у Google есть набор рекомендуемых партнерских решений, например Cloudamize, CloudPhysics и ATADATA, которые можно использовать для изучения локальной среды, сопоставления ее компонентов с соответствующими облачными сервисами и оптимальными типами инстансов, чтобы улучшить производительность.

Перейдите по ссылке cloud.google.com/solutions/migration-center, чтобы быть в курсе актуальных рекомендуемых партнерских решений для миграции.

Аналогично для фазы миграции виртуальных машин (VM) Google может порекомендовать ряд партнеров, например CloudEndure, Velostrata и ATADATA, которые опять же помогут с копированием локальных виртуальных машин непосредственно в облако. Кроме этого, Google Cloud дает возможность непосредственно импортировать виртуальные диски с помощью нативного сервиса, но это решение не совсем подходит для масштабных миграций — их стоит выполнять с помощью партнерских решений. Подробнее об аспектах миграции VM можете узнать, перейдя по ссылке cloud.google.com/compute/docs/import/.

Кроме миграции VM, в этом процессе есть еще одна важная задача, о которой стоит подумать, — масштабный перенос данных. Для ее решения существует несколько инструментов.

- *Cloud Storage Transfer Service*. Этот сервис поможет вам перенести данные из иного облачного сервиса (например, AWS) в корзины хранилища Google Cloud с помощью HTTP/HTTPS-интерфейсов. Можете сделать это с помощью консоли Google Cloud, REST API или даже клиентских API-библиотек Google Cloud. Подробнее об этом — на cloud.google.com/storage-transfer-service.
- *Google Transfer Appliance*. Как и в случае с сервисом AWS Snowball, у Google есть физическое оборудование, которое вы можете одолжить для переноса большого объема данных в облако. На текущий момент есть два варианта объема — 100 и 480 Тбайт, вы можете запросить один из них в пользование на несколько дней, чтобы подключить свои локальные системы хранилища, перенести данные, а затем переправить их в Google, в одну из своих корзин хранилища. Перед размещением на Transfer Appliance все данные подвергаются дедупликации, сжимаются и шифруются по стандартному алгоритму AES 256 с помощью заданных вами пароля и кодовой фразы. С этим сервисом вы сэкономите средства и время, так как он поможет быстро перенести данные без использования вашего собственного соединения. Подробнее об этом — на cloud.google.com/transfer-appliance/docs/4.0.

- *Google BigQuery Data Transfer Service*. В большинстве случаев клиенты используют целый ряд SaaS-приложений, например Adwords, DoubleClick Campaign Manager, DoubleClick for Publishers и YouTube. Для анализа данных с этих сервисов клиенты могут воспользоваться BigQuery Data Transfer Service, чтобы перенести свои данные непосредственно в BigQuery и создать хранилище для выполнения аналитических запросов. Этот сервис гарантирует непрерывное копирование данных в больших масштабах. Для наглядного представления тенденций на основе результатов анализа клиенты могут задействовать такие ISV-инструменты, как Tableau, Looker и ZoomData, поверх инстанса BigQuery. Подробнее об этом — на cloud.google.com/bigquery-transfer/docs/introduction.

Чтобы лучше понять, какой сервис лучше походит для миграции вашего проекта, прочтите статью Choose the Right service matrix: cloud.google.com/products/data-transfer/.

Резюме

В этой главе мы погрузились в особенности работы третьего поставщика облачных технологий, Google Cloud Platform, рассмотрели его предысторию и пути развития, познакомились с различными сервисами, такими как Cloud AI, Kubernetes Engine и G Suite. После этого рассмотрели концепции, связанные с бессерверными микросервисами, и даже создали тестовое приложение прогноза погоды, используя функции Google Cloud. Далее мы сосредоточились на автоматизации процессов и на том, как можно задействовать Google Cloud Deployment Manager для создания воспроизводимых шаблонов, предназначенных для обслуживания IaaS. Также взглянули на инструменты для реализации CI/CD-шаблонов в бессерверных окружениях и процессы контейнерного развертывания, выстроенные на основе Kubernetes. И наконец, мы изучили варианты миграции существующих локально приложений и рабочей нагрузки с помощью различных нативных сервисов Google Cloud и их партнерских решений. Всеми упомянутыми концепциями мы завершаем ознакомление с тремя наиболее известными открытыми поставщиками облачных технологий и их возможностями для создания нативных облачных архитектур приложений.

В следующей, заключительной главе мы проведем обзор всех концепций, рассмотренных в этой книге, обсудим некоторые тенденции развития технологий и, отталкиваясь от них, попробуем заглянуть в будущее, чтобы вы были готовы к непрерывным изменениям на этом поприще.

12 А что же дальше?

В предыдущих главах изучалось множество аспектов: сначала рассматривалось основное определение модели зрелости облачной среды *(Cloud Native Maturity Model, CNMM)*, затем пошагово раскрывались различные уровни, и завершалось все практической демонстрацией услуг различных облачных провайдеров. В этой главе основное внимание будет уделено целому ряду новшеств, при этом мы взглянем в будущее и обсудим наиболее актуальные прогнозы!

Нам предстоит рассмотреть следующие области.

- Семь наиболее важных прогнозов на ближайшие три года — чего следует ожидать в сфере развития архитектуры облачных приложений.
- Будущее предпринимательства в облачной среде.
- Новые перспективные ИТ-специальности (например, ИИ-директор).

Эти темы помогут:

- осмыслить перспективные тенденции развития стратегий и архитектур;
- оценить возможный рост и развитие вашей личной карьеры.

Итак, к делу!

Прогнозы на ближайшие три года — чего следует ожидать в сфере развития архитектуры облачных приложений

Хотя облачные технологии уже широко распространены во всех типах приложений и вариантах использования, но, оценивая общий потенциал рынка, можно сказать, что они пока еще не вышли из самой ранней стадии своего развития. Давайте в дополнение к уже рассмотренным тенденциям и достижениям разберем семь наибо-

лее важных прогнозов, осуществление которых серьезно поспособствует внедрению облачных технологий в ближайшие три года.

Фреймворки и платформы с открытым исходным кодом

Многие клиенты обеспокоены вынужденной привязкой, существующей в общедоступном облаке. Но их опасения совершенно напрасны, поскольку каждый компонент программного обеспечения имеет те или иные аспекты, которые зависят от конкретного поставщика и представляют ценность для клиента, поэтому рассматривать их исключительно с точки зрения привязки не стоит. К примеру, многие клиенты предпочитают работать с текстом в Microsoft Word, и если их документы созданы в нем, получается, что они в определенном смысле привязаны к данному приложению, но это еще не является признаком вынужденной привязки. Зачастую в предпринимательской деятельности используются конкретные коммерческие приложения, которые можно заменить аналогичными приложениями с открытым исходным кодом, что приведет к снижению затрат и уменьшению зависимости от конкретного поставщика. Классическим примером могут послужить многие корпоративные клиенты, в течение ряда лет пользовавшиеся базами данных Oracle, а теперь занявшиеся их перепрофилированием для работы с системами с открытым исходным кодом, такими как PostgreSQL или MySQL.

В духе данной тенденции у поставщиков облачных услуг назрела необходимость в демонстрации соответствия своих решений стандартам программного обеспечения с открытым исходным кодом, вследствие чего они также выпустили несколько программных сред и пакетов на GitHub, ответвления от которых может разрабатывать кто угодно. Общеизвестный пример такой платформы с открытым исходным кодом — Kubernetes (система с открытым исходным кодом для управления контейнерными приложениями), которая доступна по лицензии Apache-2.0 и может быть свободно загружена с GitHub (github.com/kubernetes/kubernetes).

Еще одним важным шагом в этой области, нацеленным на превращение облачных архитектур на основе микросервисов в открытые и регулируемые сообществом системы, стал запуск в 2017 году фонда облачных вычислений *Cloud Native Computing Foundation (CNCF)* (www.cncf.io/). Как было объявлено, в число самых первых платиновых участников из крупных облачных провайдеров вошли Amazon Web Services, Microsoft Azure и Google Cloud. Но со времени запуска фонда участниками сообщества стали многие организации, продолжающие поддерживать активность и открытость исходного кода программных средств, входящих в экосистему контейнерных микросервисов.

Ознакомиться с уставными документами фонда CNCF можно, перейдя по следующей ссылке: www.cncf.io/about/charter/.

Помимо участия в деятельности CNCF все поставщики общедоступных облачных сред, как правило, стараются оставаться открытыми и удобными для разработчиков. Далее приведены ссылки на специализированные сайты по тематике программных средств с открытым исходным кодом трех ведущих облачных провайдеров.

- Открытый исходный код в AWS — aws.github.io/.
- Программные средства с открытым исходным кодом в Azure — azure.microsoft.com/en-us/overview/open-source/.
- Открытый исходный код в Google Cloud — github.com/GoogleCloudPlatform.

Перспективная тенденция № 1

Число программных средств и платформ с открытым исходным кодом продолжит расти, благодаря чему облачные архитектуры станут удобнее для разработчиков.

Возросший за счет появления инфраструктурных сервисов уровень абстракции

Когда пару лет назад происходило становление общедоступных облачных услуг, основной упор делался на базовые инфраструктурные блоки, охватывающие вычислительные системы, хранилища, сети, базы данных и т. д. Но по мере внедрения различными облачными провайдерами новых сервисов произошла быстрая смена тенденции, поскольку сервисы уже представляли собой явление более высокого порядка и в значительной степени инкапсулировали в себе аспекты инфраструктуры. В качестве примера можно привести такие сервисы, как Amazon Connect, предлагаемый платформой AWS и представляющий собой облачный контакт-центр, благодаря которому конечному пользователю теперь совершенно не нужно думать о базовой инфраструктуре и он может в считаные минуты приступить к работе, сосредоточившись на основных бизнес-аспектах, логике маршрутизации вызовов и т. д.

Другая тенденция, наблюдаемая нами в контексте развертывания приложений, состоит в том, что вместо использования стандартных виртуальных облачных инстансов разработчики все чаще выбирают контейнерные или даже бессерверные способы развертывания на основе таких сервисов, как AWS Lambda. Благодаря этому пользователь еще больше абстрагируется от базовой инфраструктуры, что позволяет ему сосредоточиться на основном приложении и его бизнес-логике. Фактически эта тенденция прослеживается не только на уровне вычислений, но и при использовании баз данных (например, бессерверной Aurora от Amazon), обмене сообщениями (Amazon SQS), аналитике (Amazon Athena) и т. д.

В результате этого намечается дальнейшее изменение моделей развертывания приложений с уменьшением их зависимости от характеристик инфраструктуры, поскольку ее особенности будут автоматически учитываться самими сервисами. Еще одним сопутствующим эффектом подобных изменений станет дальнейшее раз-

мытие границ между характерными направлениями *«инфраструктура как сервис» (IaaS)*, *«платформа как сервис» (PaaS)* и *«программный продукт как сервис» (SaaS)*, и все это сведется к облачной среде, или, точнее, к сервисам, приспособленным к работе в этой среде. Признаки грядущего уже начали проступать в виде новых понятий *«функции как сервис» (FaaS)* или *«все как сервис» (XaaS)*, используемых в разных контекстах.

Перспективная тенденция № 2

С изменением модели развертывания и обретением ею независимости от базовой инфраструктуры облачные сервисы станут более ориентированными на решение прикладных задач и программное обеспечение. Кроме того, с ростом числа облачных сервисов все сильнее станут размываться границы между IaaS, PaaS и SaaS.

Системы поумнеют, станут AI/ML-управляемыми и от DevOps перейдут к NoOps

С появлением общедоступных облачных сред возникли новые типы моделей управления инфраструктурой, включающие в себя автоматическое масштабирование и методы самовосстановления и позволяющие максимально автоматизировать процессы масштабирования приложений и восстановления их работоспособности после сбоев. В ходе следующей волны автоматизации появились абстрактные облачные сервисы и стали размываться границы между чистой разработкой и эксплуатацией программных продуктов, что привело к распространению технологии DevOps. В результате мы стали свидетелями сокращения продолжительности и увеличения частоты развертывания новых версий, появления простых механизмов отката в случае возникновения проблем, а также создания более совершенных инструментов и сервисов, позволяющих упростить и унифицировать весь процесс. Но даже при использовании этих передовых методов автоматизации сохраняется потребность в эксплуатационной работе и обновлениях на конкретных уровнях операционной системы и приложений, что по-прежнему не обходится без ручного труда. Новые сервисы, к примеру Amazon Lambda и Amazon Aurora с бессерверной конфигурацией, избавляют клиентов от этих нелегких обязанностей, но еще не скоро системы станут действительно умнее и смогут самостоятельно справляться с большинством задач, связанных с эксплуатацией программных продуктов. Под этим может подразумеваться и то, что системы должны стать интеллектуальнее и вместо немедленной реакции на сложившуюся ситуацию начать заблаговременно прогнозировать возможные сценарии сбоев и выявлять необходимость любых изменений, сокращая тем самым потребность в ручной корректировке или даже создавая сложные методы автоматизации, которые могут никогда не потребоваться!

В реализации описанного перехода важная роль будет отводиться таким технологиям, как искусственный интеллект и машинное обучение. Фактически отчасти этот процесс уже пошел, поскольку услуги многих облачных провайдеров

предоставляются на основе негласных методов моделирования прогнозов. К примеру, в Amazon Macie для автоматического обнаружения, классификации и защиты конфиденциальных данных в AWS-сервисах используется машинное обучение. В Amazon GuardDuty с помощью интегрированных каналов анализа угроз выявляют злоумышленников, а для обнаружения аномалий в учетной записи и рабочей нагрузке применяется машинное обучение. Это весьма впечатляющие примеры задействования машинного обучения в сфере безопасности, но в процессе развития технологий те же принципы и механизмы получат более широкое распространение в моделях развертывания приложений и управления ими, где типовые операции претерпят существенные изменения.

Подробное описание инструментов, созданных компанией Amazon на основе математических методов и машинного обучения и предназначенных для применения мер безопасности во многих ее сервисах, можно найти в весьма интересной публикации в блоге по адресу aws.amazon.com/blogs/security/protect-sensitive-data-in-the-cloud-with-automated-reasoning-zelkova/.

Перспективная тенденция № 3

Облачные сервисы и системы станут умнее, а перечень типовых требований к инфраструктуре и эксплуатации приложений сократится, благодаря чему появятся новые принципы NoOps.

Разработчики будут сразу создавать приложения в облаке, обходясь без первоначальной локальной разработки

Несмотря на то что облачная среда приобрела широкую популярность для любого типа развертывания приложений, основная часть разработки прикладных программ (фактическое создание их программного кода) проходит в автономном режиме с использованием разработчиками IDE-среды на своих рабочих местах. Именно там они создают код, проводят его модульное тестирование и, как только добиваются успеха, перемещают код в репозиторий, который опять же может находиться не только в облачной среде, но и за ее пределами. Но после запуска процесса сборки фактическое развертывание двоичных файлов приложений происходит в основном в облаке. Главным образом это обусловлено несколькими факторами.

- Для ведения разработки в облачной среде разработчикам нужно постоянное подключение к Интернету, которое может обеспечиваться не всегда.
- IDE-среды и инструменты, к которым разработчики привыкли на своих рабочих местах, могут не предназначаться для действий в облачных средах.
- Некоторые клиенты считают, что размещать в облаке программный код — их основную интеллектуальную собственность — может быть небезопасно.

Мы уверены, что все это — лишь примеры возможных причин разработки программных продуктов в локальной среде, но основная часть этих опасений не имеет под собой ни малейших оснований. Облачная среда прошла долгий путь развития, и если разработка ведется в ней изначально, это не только упрощает сам процесс, но и позволяет разработчикам сразу же встраивать в создаваемые ими архитектуры множество облачных сервисов, так что их не потребуется программировать самостоятельно.

В качестве примера благоприятного изменения обстановки можно привести имеющееся в среде Azure облачное средство создания программного кода Visual Studio Code, а также принадлежащее компании Amazon аналогичное средство AWS Cloud9, которые позволяют заниматься программированием, используя только браузер. Следуя тем же путем, компания Microsoft приобрела сервис GitHub, главная задача которого — опять же привлечение разработчиков и облегчение разработки в облачной среде. Но это только начало, и со временем разработчикам станет намного проще создавать облачные продукты непосредственно в облачной среде, а в будущем кардинально изменится вся практика разработки и тестирования программных продуктов. Это позволит упростить глобальное сотрудничество и даст возможность создавать приложения и управлять ими децентрализованно.

Перспективная тенденция № 4

Облачные сервисы начнут заботиться о потребностях разработчиков, предоставляя им множество разных эффективных средств разработки в облаке и для облака.

На первый план выйдут модели взаимодействия с использованием голосовых команд, ботов-собеседников, а также виртуальной и дополненной реальности, работающие на основе облачных технологий

Последний год отмечен многочисленными рассуждениями и событиями вокруг искусственного интеллекта и машинного обучения (AI/ML), а также дополненной и виртуальной реальности (AR/VR). Хотя эти технологии существуют уже многие годы, для создания эффективного AI/ML-алгоритма требуется большой объем данных, а для AR/VR — изрядное количество центральных и графических процессоров. Теперь, когда для воплощения в жизнь ваших инновационных приложений облачная среда предоставляет масштабируемые базовые компоненты инфраструктуры, появилась возможность виртуально хранить петабайты данных и моментально получать тысячи центральных процессоров. Кроме этого, в данной области наблюдается существенный прогресс в развитии сервисов и вычислительных сред, подобных Apache MXNet (mxnet.apache.org/), TensorFlow (www.tensorflow.org), Caffe

(caffe.berkeleyvision.org), PyTorch (pytorch.org/) и т. д. Наметился также существенный рост количества разнообразных ботов, от простых Q&A-ботов до ботов, выполняющих конкретные задачи, например бронирование билетов и мест в отелях для туристов. Эти боты также начинают получать широкое распространение в рамках реализации облачных контакт-центров, где на основе потока маршрутизации вызовов они могут целиком выполнять множество задач, не требуя вмешательства человека. Работа таких ботов строится и на AI/ML-методах, поскольку в них неявным образом используются такие технологии, как *обработка информации на естественном языке (Natural Language Processing, NLP)*, преобразование «голос — текст» и «текст — голос», *распознавание графических образов и текста (Optical Content Recognition, OCR)*. Что же касается AR/VR-технологии, то и там наблюдается применение цифровых аватаров как для взаимодействия с конечными пользователями (к примеру, покупатели могут в интерактивном режиме примерять одежду), так и для решения разнообразных задач в производстве и предпринимательстве (например, для удаленного выявления специалистом проблем при эксплуатации сложной техники). Эти тенденции получают развитие благодаря таким облачным сервисам, как Amazon Sumerian, пригодным для создания VR-, AR- и 3D-восприятий и их запуска на таком широко востребованном оборудовании, как Oculus Go, Oculus Rift, HTC Vive Google Daydream и Lenovo Mirage, а также на мобильных устройствах под управлением Android и iOS.

Обратите внимание на следующий деморолик Google Duplex Demo от Google IO 2018: www.youtube.com/watch?v=bd1mEm2Fy08.

Еще один классический пример голосового взаимодействия — облачный голосовой сервис Amazon Alexa, допускающий прямое общение с используемыми устройствами наподобие Amazon Echo, Amazon Echo Dot или даже встраивание в ваше собственное устройство с помощью Alexa Voice Service (developer.amazon.com/alexa-voice-service). В результате такого встраивания при определенном навыке работы с Alexa можно управлять телевизором, системой отопления и освещения в доме или даже совершать банковские операции (например, проверять статус платежа по карте или инициировать платеж).

Раньше для вызова приложения пользователю нужны были доступ из браузера или мобильное приложение, а для интеграции на системном уровне пошли бы в ход и API. Но с появлением упомянутых новых механизмов взаимодействия, чат-ботов, голосового управления или интерфейсов на основе AR/VR-технологий стали изменяться и шаблоны разработки приложений. Многие организации фактически уже приступили к наработке сценариев реагирования голосового помощника Alexa, нацеленных не только на запуск приложений, но и на решение задач управления инфраструктурой — запуска инстанса EC2, получения деталей отслеживаемых событий или их удаления. А вот интерфейсы на основе жестов (имеются в виду бесконтактное движение руками или мимика) еще только зарождаются, но их широкое практическое применение уже ждет своего часа. В результате в будущем

приложения приобретут дополнительную интерактивность, усилившись множеством разнообразных интерфейсов, и получат более совершенные механизмы взаимодействия с пользователями.

Перспективная тенденция № 5

Облачные сервисы придадут приложениям дополнительную интерактивность, для чего будут задействованы управление голосом и жестами, а также виртуальная и дополненная реальность, в результате чего почти исчезнет барьер между людьми и машинами.

Облачные архитектуры выйдут за пределы центров обработки данных, распространившись и на вещи

Традиционно приложения разрабатываются либо для серверной среды, либо для мобильных устройств. Но время не стоит на месте, и теперь код и приложения создаются и для вещей. В качестве них могут выступать любые физические устройства: лампочки, термостаты, детские игрушки или даже настоящие автомобили, подключенные к Интернету и управляемые через него. В статье из публикаций компании Gartner (www.gartner.com/newsroom/id/3598917) говорится, что в 2020 году к Интернету было подключено около 20,4 млрд таких устройств. Это привело к существенным изменениям способов нашего взаимодействия с ними. Естественно, при подключении к серверной части столь большого числа устройств сильно изменяется как архитектура приложений, так и объем передаваемых и потребляемых данных. Опять же это было бы невозможно без применения облачных технологий, поскольку здесь потребуются масштабируемые методы обработки данных на основе потоков, и на первый план при реализации Интернета вещей выйдут такие сервисы, как AWS IoT и Amazon Kinesis. Кроме того, нужно учитывать и тот факт, что многие устройства не могут постоянно оставаться подключенными к Интернету из-за того, что находятся в удаленных местах, имеющих ограниченный доступ к Всемирной сети (к примеру, датчики на нефтяных вышках), поэтому для таких сценариев будут очень важны периферийные вычисления. Подойдут такие сервисы, как AWS Greengrass, позволяющие запускать локальные вычисления, обмен сообщениями и безопасное кэширование данных для подключенных устройств. Для более мелких устройств вроде микроконтроллеров, которые опять же должны поддерживать сценарии периферийных вычислений, важную роль сыграют такие новые базовые сервисы, как Amazon FreeRTOS, позволяющие выполнять подобные виды развертывания.

Поскольку данные с этих периферийных устройств будут поглощаться облаком, появится возможность проводить в режиме реального времени пакетный анализ данных или даже анализ тенденций, что опять-таки приведет к возникновению новых возможностей и развитию архитектурных моделей.

Перспективная тенденция № 6

Сквозное воздействие облачных приложений распространится от периферии до серверных приложений и порталов, ориентированных на клиентов, в результате чего изменятся способы создания и развертывания приложений.

Данные продолжат играть роль новой «нефти»

Управление данными всегда было ключевым фактором любых успешных разработки и развертывания приложений, а с появлением облака эта тенденция получила дальнейшее развитие, поскольку теперь появилась возможность использовать новые каналы информации и без проблем хранить петабайты данных. К тому же сейчас многие организации переходят к модели создания централизованных озер данных, куда последние поступают из различных источников, включая устройства, подключенные к Интернету, социальные сети и бизнес-приложения, а затем применяются для аналитики, получения более четкого представления о происходящем и выявления тенденций.

Подробности архитектуры озер данных, относящихся к AWS и Azure, можно найти по адресам aws.amazon.com/ru/solutions/implementations/data-lake-solution/ и docs.microsoft.com/en-us/azure/data-lake-store/data-lake-store-overview.

Развитию прежней тенденции поспособствовало и появление более эффективных и быстрых способов перемещения данных в облако. Еще несколько лет назад передавать данные по Интернету или сети можно было только с помощью таких сервисов, как AWS DirectConnect, или за счет использования виртуальных устройств наподобие AWS Storage Gateway. Но в последние 2–3 года все провайдеры облачных услуг стали задействовать физические устройства наподобие Amazon Snowball, которые можно заказать для своих центров обработки данных, локально подключить к передаваемым данным, а затем отправить обратно облачным провайдерам для перемещения данных в такие сервисы, как Amazon S3. Речь идет не только об этих небольших устройствах передачи данных, но и о возможности заказа перевозимого трейлером 45-футового прочного контейнера, способного передавать на каждый AWS-«снегоход» 100 Пбайт данных! И это только начало инновации, а в будущем передача данных в облако и из него станет еще проще, что поспособствует дальнейшему сбору, агрегированию и анализу огромных наборов данных. По мере перемещения данных в облако приложения станут неукоснительно придерживаться этой тенденции, благодаря чему данные продолжат играть центральную роль в повсеместном внедрении облачных технологий.

Перспективная тенденция № 7

Центральную роль в любом облачном развертывании будут играть данные, и как только способы миграции данных в облако станут проще, темпы внедрения облачной архитектуры резко возрастут.

Облачное будущее предприятий

Предприятия не любят рисковать, чем, собственно, и объясняется их нерасторопность в переходе на те или иные технологические модели. И облачные технологии, не исключение: в течение многих лет предприятия занимали выжидательную позицию, наблюдая за тем, что получится у первопроходцев. А тем временем произошел резкий всплеск множества новых стартапов, использующих облачные технологии, что послужило причиной постепенного упадка основных хорошо зарекомендовавших себя предприятий. К примеру, компании Lyft и Uber бросили вызов таксомоторному и транспортному бизнесу, компания Airbnb внесла сумятицу в индустрию аренды жилья, а компания Oscar Insurance радикально изменила сектор медицинского страхования. Ранее такие резкие взлеты никогда не удавались, но теперь, при наличии облачных технологий, каждый получил доступ к одному и тому же набору сервисов и ресурсов инфраструктуры, масштабируемых в ту или иную сторону в зависимости от текущих потребностей бизнеса... и это изменило правила игры для всех.

Из-за упомянутого эффекта руководство многих предприятий осознало, что прежде, чем кто-то другой растормошит их бизнес, они сами должны сделать это и взойти на гребень волны облачных инноваций. И следует отметить, что многие предприятия не только приступили к внедрению облачных технологий, но в последние годы даже пошли ва-банк, предприняв попытку пересмотреть всю свою рабочую и бизнес-модель. Перечень корпоративных клиентов постоянно растет, и в поле зрения попали даже такие гиганты, как GE, Capital One, Adobe, Hess, Kellogg's, Novartis, Infor, Suncorp, BestBuy, Philips, Goldman Sachs и т. д., обнародовавшие информацию об использовании той или иной облачной платформы.

Конкретные примеры от различных облачных провайдеров можно найти по следующим ссылкам:

- AWS: aws.amazon.com/solutions/case-studies/;
- Azure: azure.microsoft.com/en-us/case-studies/;
- Google Cloud: cloud.google.com/customers/.

Активное внедрение облачных технологий прослеживается не только в коммерческих, но и в государственных и общественных организациях. В качестве примеров можно привести NASA, FDA, FINRA, министерство национальной безопасности США, правительство Сингапура, корпорацию Transport for London, правительство

Онтарио, совет Business Sweden, город Тель-Авив, министерство здравоохранения Чили и т. д. У них у всех имеются те или иные приложения, развернутые в облаке и доступные по открытым ссылкам на веб-сайтах различных облачных провайдеров.

В жизни предприятий произошли существенные изменения, ведь это не просто технологическое обновление или переход на другую платформу, но и нововведения, влекущие за собой многочисленные эксплуатационные и бизнес-последствия, поскольку теперь все ориентировано на спрос или на соотношение операционных расходов с капитальными затратами, что для этих предприятий стало принципиально иной моделью. Кроме того, чтобы набрать высокий темп и получить максимальный эффект от облачных технологий, им теперь необходимо иметь небольшие команды (например, DevOps-команды, довольствующиеся для перекуса двумя пиццами), способные быстро опробовать новые облачные сервисы и функции и включить их в свои прикладные архитектуры. Помимо следования принципам гибкости и инновационности, предприятиям необходимы также надлежащее управление мерами безопасности и контроль за их соблюдением, чтобы можно было продемонстрировать их аудиторам и обеспечить строгое соблюдение законов страны. Чтобы изменения происходили размеренно, различные провайдеры облачных услуг разработали комплексные среды управления внедрением облачных технологий, помогающие ИТ- и техническим директорам корпоративных клиентов и другим заинтересованным сторонам переводить решения своих задач в облако без проблем и сбоев.

Интересную информацию на эту тему можно найти в блоге AWS Enterprise Strategy: aws.amazon.com/blogs/enterprise-strategy/.

Изменения ставят перед корпоративными клиентами еще одну задачу — обучение персонала работе с избранной облачной платформой. Для этого у всех облачных провайдеров есть комплексные программы обучения и сертификации, призванные помочь клиентам и их сотрудникам освоить технические возможности сервиса, методы встраивания в облачную среду и эффективные приемы работы в облаке. Но, помимо обученных сотрудников, корпоративным клиентам нужны специалисты, способные поддержать их в случае затруднений при реализации проекта. Для этого у всех облачных провайдеров имеются профессиональные консультанты (Professional Services consultants), способные прийти на помощь и проинформировать клиентов по различным техническим и технологическим аспектам, а также организационным вопросам реализации проекта. Для дальнейшего содействия в создании центров оказания компетентной помощи по работе в облачной среде (CCOE) и предоставления команде разработчиков необходимых знаний об облачных технологиях клиенты в большинстве случаев пользуются услугами различных консалтинговых фирм, которые специализируются в разных областях и помогают делать облачные проекты успешными.

С помощью упомянутых составляющих клиенты могут изменить свою повседневную деятельность и добиться более плавного и успешного перевода всех своих

задач в облачную среду. В результате практической реализации облачных проектов мы стали свидетелями появления весьма интересных инноваций, исходящих от корпоративных клиентов, среди которых:

- сценарии реагирования голосового помощника Alexa от Amazon в Capital One (www.capitalone.com/applications/alexa/);
- служба регулирования отрасли финансовых услуг FINRA, ежедневно с помощью AWS собирающая и анализирующая миллиарды записей о брокерских трансакциях (youtu.be/rHUQQzYoRtE);
- презентация лаборатории реактивного движения NASA, которая посредством облачных технологий помогает отвечать на вопросы о космосе (youtu.be/8UQfrQNo2nE);
- опыт компании BMW в использовании облачного социального маркетинга для поддержки запуска моделей и выстраивании перспектив (customers.microsoft.com/en-us/story/bmw-supports-model-launch-develops-prospects-with-clo2);
- опыт применения подразделением Airbus, отвечающим за оборонную и аэрокосмическую продукцию, машинного обучения в облаке для повышения качества спутниковых снимков (cloud.google.com/blog/big-data/2016/09/google-cloud-machine-learning-now-open-to-all-with-new-professional-services-and-education-programs).

Новые специальности в сфере информационных технологий

Одно из долгосрочных последствий внедрения облачных технологий в деятельность предприятий — появление множества новых специальностей. Некоторые из них уже получили широкое распространение, а востребованность остальных возрастает по мере развития технологических тенденций.

- *Ведущий инженер по технологиям и инновациям (Chief Technology & Innovation Officer, CTIO)*. В прежние времена в организациях был либо ИТ-, либо технический директор, а теперь благодаря повышенному вниманию к инновациям в основном под влиянием внедрения облачных технологий появляется специальность CTIO.
- *Архитектор облачных решений (Cloud Solutions Architect)*. Прежде привычными были специальности архитекторов приложений, систем, интеграций и т. д., а с появлением возможностей, предоставляемых облачными архитектурами, весьма актуальной стала специальность архитектора облачных решений.
- *Архитектор миграций в облачную среду (Cloud Migration Architect)*. Поскольку, прежде чем начать эффективно использовать облачные технологии, многим организациям предстоит погасить технический долг огромного масштаба, сравнительно недавно появилась специальность архитектора миграций в облачную среду, в чьи обязанности входит решение вопросов новых миграций.

- *Инженер по разработке и эксплуатации программных средств, а также автоматизации облачных технологий (DevOps Professional/Cloud Automation Engineer)*. Для полноценного применения возможностей облачной среды нужно сконцентрироваться на автоматизации и согласовании облачных функций, где зачастую не обойтись без создания программ, сценариев и надлежащих рабочих процедур. Поэтому ключевое значение стала приобретать специальность инженера по разработке и эксплуатации программных средств, а также автоматизации облачных технологий.
- *Архитектор облачной безопасности (Cloud Security Architect)*. Меры безопасности в облаке и контроль за их соблюдением сильно отличаются от таковых в локальной установке. Поэтому для грамотного использования средств обеспечения безопасности в облачной среде с применением надлежащих моделей управления ими и обеспечения соответствия существующим требованиям во многих организациях вводят должность архитектора облачной безопасности.
- *Специалист по облачной экономике (Cloud Economics Professional)*. Эта уникальная специальность еще не получила широкого распространения, но некоторые организации уже начали нанимать профессионалов, призванных оптимизировать применение облака с точки зрения экономики. Они заняты главным образом поиском наиболее удачных финансовых конструкций (например, зарезервированных инстансов виртуальных машин) или даже оказанием помощи в разделении затрат внутри организации с использованием тегирования.

Наряду с перечисленными существуют и другие специальности, находящие широкое применение в результате роста объема так называемых больших данных, аналитики и машинного обучения, подстегиваемого развитием облачных технологий. При прогнозируемых в ближайшие годы темпах развития технологий искусственного интеллекта и машинного обучения, возможно, будут востребованы новые, соответствующие этой области специальности. Одна из них — ИИ-директор (Chief AI Officer), нацеленная не только на технологические аспекты достижения наивысшей эффективности применения AI/ML-технологии, но и на отслеживание возникающих в результате ее внедрения социальных последствий. К примеру, когда развивающаяся AI/ML-технология достигнет уровня самостоятельного принятия решений, позволяющего, скажем, управлять автомобилем, то кто будет нести ответственность за сбитого беспилотником пешехода — искусственный интеллект или создавший его разработчик? Вопрос это непростой, именно поэтому для разрешения новых ситуаций, оказывающих влияние на общество, и даже для работы с правительственными и различными регулирующими органами с целью создания надлежащих правовых рамок для поддержания баланса общественных отношений и будут вводиться такие новые специальности, как ИИ-директор.

В общем, на фоне упомянутых изменений для успешной и конкурентоспособной деятельности в наступившую эру инноваций предприятиям придется развиваться и адаптироваться!

Резюме

Вот мы и подошли к последнему разделу книги, накопив с самых первых страниц весьма солидный объем информации. Давайте вернемся к началу и осмыслим, какие знания нам удалось почерпнуть, изучив целый ряд глав.

Все начиналось с определения понятия «облачный», поскольку именно оно легло в основу дальнейшего повествования. Итак, кратко напомним, что модель зрелости облачной среды (CNMM) вращается вокруг трех основных осей:

- облачных сервисов;
- программно-ориентированной разработки;
- автоматизации.

То есть в распоряжении каждого клиента окажутся все эти оси различной степени зрелости, но по сути своей все они могут быть облачными (рис. 12.1).

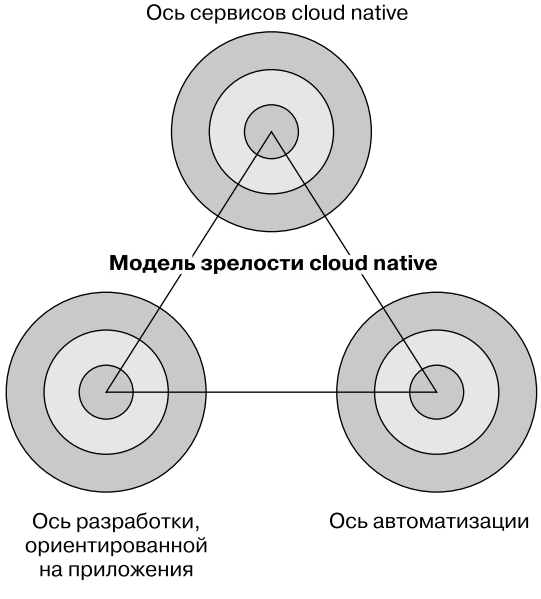


Рис. 12.1

Затем мы перешли к особенностям пакета Cloud Adoption Framework и рассмотрели его значение с различных точек зрения: бизнеса, простых людей, методов корпоративного управления, платформы, безопасности и обычной деятельности компании. В итоге все это привело нас к следующему важному набору тем, вращающихся вокруг сущности микросервисов, бессерверных компонентов и методов

создания приложений в облаке с помощью 12-факторной среды приложений. Затем была рассмотрена облачная экосистема, состоящая из технологических и консультационных партнеров, различных моделей лицензирования и получения программных средств, включая выбор торговых площадок, использование собственных лицензий и т. д. После усвоения всего многообразия этих ключевых понятий мы начали постепенно углубляться в подробности того, как следует рассуждать о масштабируемости, доступности, безопасности, управлении затратами и производственной эффективности, играющих в облачных технологиях весьма важную роль, а также рассмотрели их сходство с аналогичными категориями существующих локальных моделей и отличия от них.

Когда все эти понятия были усвоены, настало время засучив рукава всерьез взяться за особенности систем, предоставляемых тремя наиболее известными облачными провайдерами: Amazon WebServices, Microsoft Azure и Google Cloud Platform. В каждой из посвященных им глав рассматривались их возможности и различия на основе ранее изложенной CNMM-модели. Были изучены основные сервисы, предлагаемые каждым облачным провайдером, проецирующиеся на ось 1 CNMM-модели, а затем вопросы разработки и развертывания бессерверных микросервисов, проецирующиеся на ось 2 CNMM-модели, и в завершение — возможности автоматизации и DevOps-технологии, проецирующиеся на ось 3 CNMM-модели. Все это дало возможность усвоить понятия, используемые всеми облачными провайдерами, и провести их сравнительный анализ.

Основной линией повествования в последней главе стали семь наиболее ярких технологических трендов, развитие которых ожидается в ближайшие годы. Это поможет не только взглянуть на имеющиеся возможности, но и выстроить планы на будущее. После этого мы углубились в вопросы влияния облачных технологий на предпринимательскую деятельность и восприятие произошедших изменений предпринимателями. В том же разделе были изучены новые ИТ-специальности, изменяющиеся по мере того, как облачные технологии получают все более широкое распространение в бизнесе.

Со всем этим багажом мы пришли к заключению, что для вас главным из всего здесь изложенного должна стать следующая мысль: *«Облачные технологии — веяние времени, и чтобы добиться максимальной эффективности от их использования, на переход к ним нужно положить все свои силы!»*