1 Что такое APIOps

В этой главе

- ✓ Проблемы, связанные с масштабированием разработки и доставки API
- ✓ APIOps как комбинация DevOps и GitOps для разработки API
- ✓ Принципы APIOps
- ✓ Преимущества и недостатки APIOps на основе OpenAPI

Во многих организациях стремятся разрабатывать качественные, хорошо документированные и простые в использовании API, однако достичь этого на практике бывает совсем непросто. Порой не помогают ни четкие стандарты и функциональные возможности управления, ни подход на основе проектирования — итоговый продукт все равно разительно отличается от задуманного. Реализованные API зачастую не соответствуют требованиям к удобству и безопасности, имеют проблемы с документацией, а пользователям трудно найти их среди других ресурсов и получить к ним доступ. Более того, могут возникать проблемы с производительностью, надежностью и масштабируемостью. Разработка API может обернуться весомыми затратами и оказаться неэффективной, из-за чего продукт появится на рынке с большим опозданием.

Решением проблемы служит APIOps. Этот подход помогает автоматизировать доставку API и тем самым повысить эффективность его проектирования, разработки, тестирования и развертывания. Автоматизация APIOps затрагивает управление файлами определения и конфигурации API и внедряется в пайплайн непрерывной интеграции (Continuous Integration, CI) и непрерывной доставки (Continuous Delivery, CD). APIOps обеспечивает сквозную автоматизацию жизненного цикла API, опираясь на принципы DevOps и GitOps, о которых я расскажу позже. С помощью этого подхода в компаниях разрабатывают более качественные и удобные API. На этапе проектирования традиционные методы управления, при которых человек проверяет API вручную, уступают место проверкам с помощью программных средств, включая линтинг и выявление критических изменений. Кроме того, повышается надежность релизов API, так как конфигурации развертываются через Git. API становятся более контролируемыми и лучше соответствуют требованиям, а команды разработки и эксплуатации взаимодействуют эффективнее. Таким образом благодаря тому, что в APIOps соблюдение стандартов и правил контролируется автоматически, организации могут ускорить доставку и развертывание API, а также гарантировать их согласованность. Рассмотрим это на примере.

1.1. Масштабирование доставки API: возможные проблемы

Представьте вымышленную компанию электронной коммерции, например книжный интернет-магазин BookInc. Ее ключевой продукт — облачная платформа BookStore API, которая позволяет партнерам продавать книги от имени BookInc.

На ранних этапах работы с API инженеры BookInc имели полную свободу в проектировании и развертывании изменений в REST API BookStore. Они не были связаны строгими ограничениями и поэтому могли быстро выпускать обновления. Однако с ростом компании появилось больше команд, которые вносили изменения в BookStore API, и в результате он начал работать нестабильно. Пользователи жаловались, что одни и те же ресурсы назывались по-разному в зависимости от раздела API, а сообщения об ошибках не имели единого формата. Кроме того, отсутствовала справочная документация, некоторые эндпоинты были не защищены, а другие постоянно возвращали ошибки.

Чтобы устранить эти проблемы, в компании BookInc сформировали централизованную команду для работы с изменениями BookStore API. Она разрабатывала стандарты API и следила за тем, чтобы им соответствовали все изменения, а также отвечала за работу API-шлюза и развертывание на нем API. Помимо этого, члены команды писали функциональные тесты, чтобы убедиться, что API работает в точности так, как указано в опубликованной документации.

В рамках управления проектированием BookStore API команда платформы вручную проверяла все предложенные изменения, чтобы убедиться, что они соответствуют стандартам. Она также обучала оптимальным методам разработки, закрепленным в стандартах API, и подчеркивала, как важно их соблюдать. Чтобы конечный продукт удовлетворял запросам пользователей, команда API-платформы призывала к новому подходу, при котором в первую очередь выполняется проектирование API (*design-first*): сначала разработчики должны создать спецификацию OpenAPI (OpenAPI Specification, OAS), пройти проверку и получить одобрение, а уже потом приступить к реализации бэкенда.

Поначалу все было хорошо. Команда API-платформы внедряла качественные решения, руководствуясь лучшими намерениями. Однако с ростом BookInc и появлением новых команд разработчиков возникло несколько проблем.

- Когда команда, разрабатывающая какую-либо функцию, хотела согласовать изменения в API, ей приходилось создавать заявки или назначать встречи со специалистами API-платформы. Так возникали задержки: несколько команд были вынуждены ждать, пока запросы попадут в план, пройдут ревью и получат одобрение. В итоге ожидание затягивалось, и на развертывание изменений в API уходило больше времени.
- Так как команда API-платформы вручную проверяла, соответствуют ли изменения стандарту, она действительно повысила согласованность API и соблюдение требований. Однако человеческий фактор внес свои коррективы: иногда возникали ошибки. Некоторые серьезные проблемы обнаруживались уже после того, как изменения попадали на продакшен и клиенты начинали использовать новую версию.
- Возросло количество развертываний API, увеличилось время, необходимое для создания и поддержки автоматизированных тестов, и нагрузка на централизованную команду API-платформы многократно повысилась. Чтобы успеть выполнять другие задачи, специалисты стали проводить ревью только один раз в неделю.
- Ревью отнимали слишком много времени, и под натиском сжатых сроков некоторые команды разработчиков отказывались от подхода *design-first* и начинали писать бэкенд-микросервисы, не дожидаясь, пока команда платформы рассмотрит предложенные изменения. Но иногда из-за этого им приходилось значительно переделывать выполненную работу, поскольку утвержденный дизайн сильно отличался от их реализации, что не могло не разочаровывать.
- Разработчики стали прилагать больше умственных усилий: чтобы не нарушать требования, им приходилось запоминать все стандарты API и следить за их обновлением.

Вот так компания BookInc изменила подход к управлению API. Отказавшись от распределенной системы, ориентированной на скорость, она перешла

к централизованной модели с упором на качество и надежность. Однако за это пришлось заплатить временем и снизить темпы разработки. Кроме того, процесс развертывания изменений в API стал неудобным для разработчиков: при новом подходе нужно создавать заявки, отправлять электронные письма и назначать встречи, а затем долго ждать своей очереди и медленно проходить все этапы обратной связи. Внутри компании вырос уровень напряжения и раздраженности, что плохо сказалось на корпоративной культуре.

1.2. Решение проблем, вызванных масштабированием

В нашем примере растущая компания BookInc должна сохранить баланс между качеством и скоростью выпуска API. К счастью, существует проверенный подход к разработке ПО, который позволяет справиться с такой задачей, — DevOps. Любой, кто с ним знаком, понимает, что он помогает устранить конфликты между разработчиками и командами, которые поддерживают платформу и инфраструктуру API. DevOps опирается на принципы бережливого производства (lean-принципы) и охватывает весь процесс создания ценности — от разработки до эксплуатации, — чтобы вы могли быстро и надежно доставлять ее клиентам. Одно из ключевых преимуществ подхода — ускорение работы при одновременном снижении количества отказов, то есть при повышении качества. Вы увидите, как это помогает оптимизировать процессы разработки BookInc API.

В книге «The DevOps Handbook»[1] (2-е изд., IT Revolution Press, 2021) Дж. Ким (G. Kim) выделяет три принципа DevOps.

1. Создание потока ценности с быстрым течением слева направо (от разработки к эксплуатации).
2. Налаживание потока обратной связи (с течением справа налево), который позволяет оперативно выявлять и устранять проблемы.
3. Формирование благоприятной среды для постоянного обучения, обмена знаниями и экспериментов, которая способствует доверию внутри команды.

Эти принципы ложатся в основу методов DevOps, включающих следующее.

- *Контроль версий* — использование систем для отслеживания ревизий и истории изменений кода и файлов конфигурации. Позволяет просматривать правки, восстанавливать предыдущие версии и исправлять ошибки. С помощью такой системы несколько разработчиков могут писать код совместно, сохраняя всю историю изменений. В качестве примера популярного инструмента для контроля версий можно назвать Git.

[1] Дж. Ким и др. «Руководство по DevOps. Как добиться гибкости, надежности и безопасности мирового уровня в технологических компаниях».

- *Непрерывная интеграция и непрерывная доставка* (CI/CD) — подход, при котором разработчики несколько раз в день интегрируют протестированный код в общую ветку репозитория. Системы непрерывной интеграции (или билд-серверы) автоматически собирают и упаковывают код в готовые артефакты, благодаря чему возможно их частое развертывание. Следующий этап — непрерывная доставка, во время которой артефакты автоматически развертываются в тестовой и продакшен-среде (иногда в нескольких). Благодаря CI/CD разработчики всегда готовы развернуть артефакты, которые прошли стандартный процесс тестирования.
- *Инфраструктура как код* (Infrastructure as Code, IaC) — способ определения системных ресурсов и топологии, например сетей, виртуальных машин и балансировщиков нагрузки (а также управления ими), с помощью описательной модели. Он позволяет изменять и хранить определения ресурсов в системе контроля версий так же, как и программный код. Благодаря таким инструментам IaC, как Terraform, Ansible или AWS, можно автоматизировать развертывание инфраструктуры, избегая ручных процессов и предотвращая дрейф окружения. IaC обеспечивает воспроизводимость и стабильность среды при каждом новом развертывании.
- *Конфигурация как код* (Configuration as Code, CaC) — подход к управлению конфигурацией приложения с помощью описательной модели, которая хранится в системе контроля версий. В отличие от подхода IaC, касающегося инфраструктуры, CaC относится к уже развернутым приложениям. Он позволяет командам создавать паттерны конфигураций, внедрять их и отслеживать изменения, предотвращая дрейф конфигурации.
- *Непрерывный мониторинг и измерения* — сбор, анализ и передача данных о состоянии системы на продакшен, благодаря которым разработчики могут ее улучшать. Этот принцип охватывает два параметра: мониторинг и наблюдаемость. *Мониторинг* включает в себя настройку метрик, оповещений и дашбордов, которые позволяют командам отслеживать состояние систем и оперативно реагировать на заранее выбранные изменения параметров. *Наблюдаемость* — это возможность запрашивать и анализировать информацию о поведении инструментальных систем и сред, чтобы выявить первопричину проблем, даже если их не удалось предвидеть. Инструменты мониторинга и наблюдаемости позволяют командам отслеживать общую производительность и состояние системы в режиме реального времени. Это достигается за счет сбора телеметрических данных, включая информацию о событиях и журналы работающих систем, — их можно анализировать и запрашивать по мере необходимости.
- *Обучение на ошибках* — в сложных программных системах неизбежно случаются сбои. Однако из них можно извлекать ценные уроки, беспристрастно анализируя инциденты и экспериментируя: иногда специалисты намеренно создают сбои, чтобы понять, как системы ведут себя в нестабильных условиях.

Такая практика называется «хаос-инжиниринг». Команды устраивают «игры», в ходе которых имитируют отказ системы и проверяют свою готовность к инцидентам. Кроме того, они организуют специальные учения для тестирования безопасности, где некоторые специалисты играют роль злоумышленников из «красной команды», пытаясь осуществить взлом.

DevOps также охватывает другие принципы работы с продуктами, процессами и корпоративной культурой, включая сбор и внедрение обратной связи от пользователей, работу небольшими итерациями, упрощенные процессы утверждения, постоянное тестирование, поддержку экспериментов и взаимодействия в команде. Эффективность DevOps измеряется четырьмя основными метриками: частотой развертывания (deployment frequency), временем реализации изменений (lead time for changes), долей отказов в результате изменений (change failure rate) и средним временем восстановления (mean time to restore). Более подробно подходы и метрики DevOps описали Николь Форсгрен, Джез Хамбл и Джин Ким (Nicole Forsgren, Jez Humble, Gene Kim) в книге «Accelerate»[1] (IT Revolution, 2018).

Вернемся к компании BookInc, которая выбрала трудоемкий процесс утверждения изменений в API, и рассмотрим альтернативный, более удобный для разработчиков подход к доставке ПО, решающий возникшие проблемы, — GitOps. С его помощью можно управлять изменениями в облачных приложениях и инфраструктуре. В GitOps конфигурация системы хранится и обновляется через Git. Пайплайны доставки автоматически применяют все изменения, зафиксированные в Git, к нужным средам. При этом ручное редактирование конфигурации через пользовательский интерфейс исключено, и инженеры вносят изменения только через пул-реквесты.

Одно из преимуществ GitOps в том, что он поддерживает принцип самообслуживания. В традиционном подходе разработчики отправляют команде эксплуатации тикеты на изменение инфраструктуры, однако с появлением новых специалистов обработка тикетов становится узким местом, и централизация замедляет общую работу. Модель GitOps решает эту проблему: разработчики сами изменяют конфигурацию, отправляя пул-реквесты, которые затем проверяет и утверждает команда эксплуатации. Далее процесс развертывания в нужной среде выполняется автоматически. Такой подход гораздо эффективнее и удобнее для разработчиков, поскольку основан на Git — инструменте, с которым большинство из них уже знакомы. GitOps помогает найти баланс между скоростью разработки и контролем изменений в инфраструктуре и конфигурации. А отменить обновление можно всего одной командой `git revert`.

---

[1] Н. Форсгрен, Дж. Хамбл и Дж. Ким. «Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации».

Еще одно важное преимущество GitOps — встроенные механизмы для тестирования на соответствие и возможность аудита. Все файлы конфигурации хранятся в системе контроля версий, что позволяет отслеживать, кто, когда и какие изменения внес. Специалисты также могут автоматически проверять, соответствуют ли файлы стандартам и требованиям организации, а затем формировать об этом отчеты.

Рабочая группа GitOps (https://opengitops.dev) отмечает, что в основе этого подхода лежат четыре принципа. Состояние системы (совокупность всех параметров конфигурации, необходимых для воссоздания системы) должно соответствовать следующим требованиям.

- *Декларативность*. Для каждой среды нужно описать желаемое состояние программной системы — конфигурацию приложения и инфраструктуру — в декларативной форме. При этом не указывается, как именно оно будет достигнуто. Состояние задается не набором команд или пошаговых инструкций, а в виде конечной конфигурации.
- *Версионность и неизменяемость*. Желаемое состояние хранится в системе контроля версий, например Git. Так обеспечивается его неизменяемость и сохраняется история версий.
- *Автоматическое извлечение*. Программные агенты автоматически извлекают желаемое состояние системы из выбранного источника, сравнивают его с развернутой конфигурацией и предупреждают в случае расхождения. Они обеспечивают цикл управления и обратной связи, что позволяет системе восстанавливаться самостоятельно.
- *Непрерывное согласование*. Обнаружив расхождения между желаемым и фактическим состоянием системы, программные агенты автоматически вносят корректировки. Такое случается, если специалисты изменяют состояния вручную. Какой бы ни была причина расхождения, программные агенты его выявляют и изменяют фактическое состояние так, чтобы оно соответствовало желаемому.

Более подробно о принципах GitOps в контексте облачной среды рассказали Билли Юэн, Александр Матюшенцев, Тодд Экенстам и Джесси Суэн (Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, Jesse Suen) в книге «GitOps and Kubernetes» (Manning, 2021; https://mng.bz/Bdq0).

Интеграция описанных принципов DevOps и GitOps в процессы разработки и доставки API формирует концепцию APIOps — сквозную автоматизацию жизненного цикла API. Именно поэтому я упоминал, что APIOps повышает эффективность проектирования и доставки API за счет автоматизации всех этапов работы. В этой книге я рассматриваю APIOps в контексте продуктов API, разрабатываемых по принципам REST (REST API) и применяемых в микросервисных архитектурах. Для определения этих интерфейсов используется OpenAPI.

Применим ли APIOps к другим форматам спецификаций API?

Спецификация OpenAPI (OpenAPI Specification, OAS; www.openapis.org), ранее известная как Swagger, представляет собой язык описания интерфейсов (Interface Definition Language, IDL), предназначенный для определения и документирования API, работающих по протоколу HTTP (HTTP API). Однако OpenAPI — далеко не единственный формат, который применяется в разработке API. Рассмотрим несколько альтернатив.

- *RAML* (RESTful API Modeling Language, язык моделирования REST API; https://raml.org) — еще один язык описания интерфейсов для HTTP API. API, определенные с использованием RAML, создаются в формате YAML.
- *API Blueprint* (https://apiblueprint.org) — язык определения API, основанный на синтаксисе Markdown.
- *AsyncAPI* (www.asyncapi.com) — спецификация, созданная по образцу OpenAPI. Этот формат описания интерфейсов служит для определения и документирования API, которые работают на основе обмена сообщениями. Он не привязан к конкретному протоколу обмена сообщениями и поддерживает в том числе AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport), WebSockets, Kafka и STOMP (Simple Text Oriented Messaging Protocol).
- *gRPC* (https://grpc.io/) — высокопроизводительный фреймворк для удаленного вызова процедур (Remote Procedure Call, RCP), с помощью которого приложения и сервисы обмениваются информацией. Обычно gRPC применяется вместе с Protocol Buffers (Protobuf) — протоколом сериализации структурированных данных, не зависящим от языка программирования.

К некоторым из перечисленных форматов API, безусловно, применимы отдельные принципы APIOps, рассмотренные в этой книге. Например, с помощью инструмента для линтинга API, который мы рассмотрим в главе 2, можно автоматически проверять, соответствуют ли файлы определения AsyncAPI руководству по стилю. Однако, несмотря на возможность применить APIOps к другим форматам, подробный разбор этой темы выходит за рамки моей книги.

Я расскажу, как применять принципы APIOps на каждом этапе жизненного цикла API: автоматизировать управление файлами определения API, тестировать их на соответствие и настраивать конфигурацию API-шлюзов. Такой подход позволит ускорить доставку API клиентам, сократив время на реализацию, и при этом соблюсти стандарты управления и качества (рис. 1.1).

Итак, APIOps предполагает полную автоматизацию жизненного цикла API, но для начала важно понять, что такое продукт API и чем он отличается от интеграционного сервиса API.

1.3. API и продукты API

Программный интерфейс приложений (Application Programming Interface, API) — это набор функций и протоколов, с помощью которых программные системы взаимодействуют друг с другом. Он служит своего рода границей, через которую

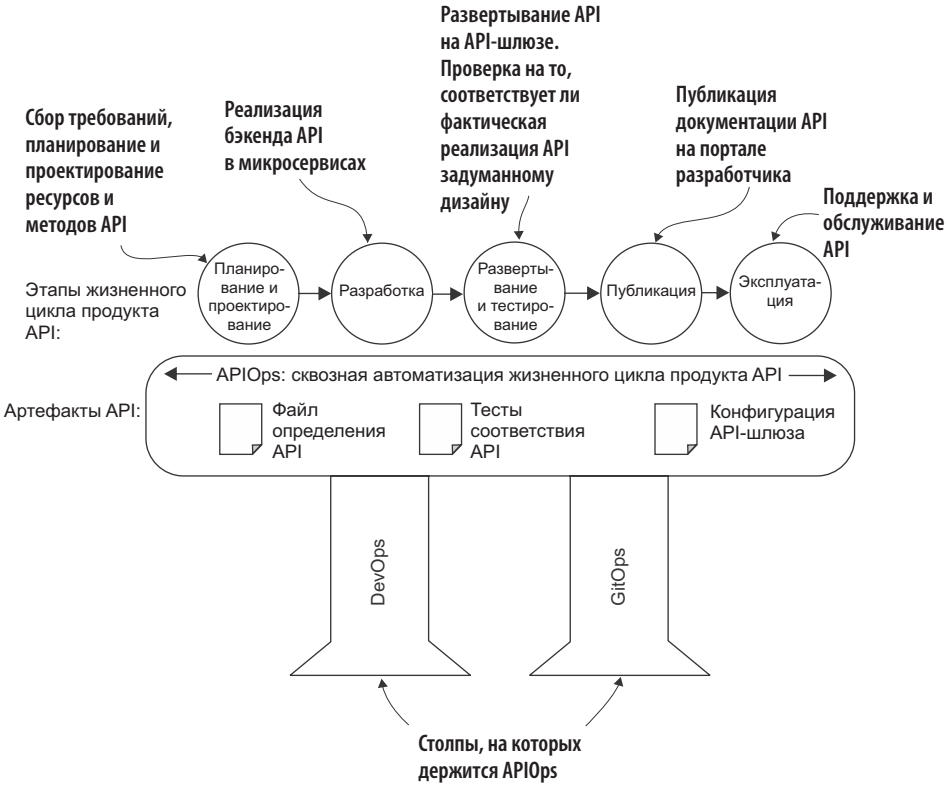

Рис. 1.1. APOps обеспечивает сквозное управление жизненным циклом API, позволяя ускорить поток создания ценности и при этом сохранить высокие стандарты управления и качества

происходит передача информации. Существуют различные типы API-решений, например REST API, SOAP API, RPC API, GraphQL API и WebSocket API. В свою очередь, *продукт API* — полноценное пакетное API-решение, ориентированное на конкретные задачи потребителя. Оно предлагает надежную систему безопасности, надлежащую документацию и каналы поддержки. За продукт API отвечает специальная команда, которая контролирует весь его жизненный цикл, от идеи до завершения поддержки. Разработчики должны понимать потребности пользователей API, улучшать взаимодействие с продуктом и следить за достижением основных метрик компании. Если продукт API предлагается на платной основе, необходимо продумать модель ценообразования и подробно объяснить, как рассчитывается и взимается плата за использование услуги. Кроме того, у продукта API должно быть соглашение об уровне обслуживания, в котором прописаны параметры качества, например время безотказной работы и ограничение частоты запросов. Девять ключевых компонентов продукта API представлены на рис. 1.2.

Рис. 1.2. Продукт API — комплексное решение, созданное для удовлетворения потребностей пользователя API. Он состоит из девяти компонентов, включая надлежащую документацию, каналы поддержки, а также команду, которая отвечает за его жизненный цикл

Продукты API делятся на два типа: внешние API, реализующие бизнес-функции, и внутренние, предназначенные для технического взаимодействия (например, внутренний API для управления правами доступа). API становится продуктом, когда им управляют как продуктом. Сначала разработчики выявляют и анализируют проблемы потребителей и определяют, какую пользу может принести API. Затем команда создает и утверждает дизайн API для удовлетворения потребностей, внедряет его, документирует и публикует готовое решение. Но на этом работа не заканчивается: специалисты постоянно отслеживают состояние API, анализируя отзывы пользователей и собираемые данные, и улучшают продукт на протяжении всего его жизненного цикла.

В результате API разрабатывается как самостоятельный продукт и, в отличие от обычных API-решений, требует больше ресурсов на создание документации, поддержку и заботу об удобстве пользователей. Принципы и методы APIOps, рассматриваемые к этой книге, направлены именно на разработку продуктов API.

В управлении продуктами используется модель жизненного цикла, которая помогает анализировать состояние продукта и принимать связанные с ним решения. Она описывает последовательные этапы существования продукта, включая разработку, рост, зрелость и спад. Зная их, менеджеры могут продумывать и обсуждать планирование, реализацию и развитие решений. Продукты

API проходят тот же самый цикл — от идеи до завершения поддержки. Чтобы эффективнее планировать, разрабатывать и поддерживать API, специалисты команды должны использовать единые термины для описания жизненного цикла. Однако каждая организация определяет его по-своему. В этой книге рассматривается упрощенная модель.

1. *Планирование и проектирование*. На этом этапе рождаются и фиксируются новые идеи для API. Определяются и анализируются функциональные и нефункциональные требования, создается концептуальная модель предметной области. Затем требования преобразуют в формальное определение API, которое компания обсуждает и утверждает с потенциальными пользователями.
2. *Разработка*. API реализуется как часть бэкенд-микросервисов. Этап включает написание кода, юнит-тестирование и упаковку. Кроме того, создается имитация (мок) API, что позволяет параллельно разрабатывать сервисы, которые будут использовать этот интерфейс.
3. *Развертывание и тестирование*. Микросервисы настраиваются в API-шлюзе, после чего API развертывается во вспомогательной среде для приемочного тестирования.
4. *Публикация*. На портале разработчика размещают документацию по API и примечания к релизу.
5. *Эксплуатация*. API развертывают и запускают на продакшене. Команда постоянно отслеживает его состояние на предмет проблем с функциональностью, производительностью и безопасностью.

Любая концептуальная модель упрощает сложные процессы, чтобы подчеркнуть в них определенные аспекты, — и эта модель не исключение. С ее помощью я хотел бы объяснить методы APIOps и подходы к управлению продуктами API на основных этапах жизненного цикла. Однако она не описывает все этапы проектирования и доставки API и не углубляется в технические подробности, поскольку обычно рабочий процесс бывает более гибким и итеративным. Если вы хотите лучше изучить жизненный цикл API, рекомендую книгу Луиса Вейра (Luis Weir) «Enterprise API Management» (Packt Publishing, 2019).

1.4. Зачем внедрять APIOps?

APIOps позволяет автоматически развертывать продукты API и управлять ими. Однако прежде чем внедрять APIOps, важно разобраться в существующем процессе создания ценности: понять ключевые этапы и действия на пути от запроса клиента до развертывания API в продакшен-среде. Эти знания помогут определить приоритетные области, выигрывающие от автоматизации. Подробнее о том, как планировать проектирование и доставку API, я расскажу в главе 2.