O'REILLY®

Kubernetes на практике

Создание успешных платформ приложений

Джош Россо, Рич Ландер,
Александр Бранд, Джон Харрис

bhv®

УДК 004.273
ББК 32.973-018.2
Р77

Россо, Д.

Р77 Kubernetes на практике: Пер. с англ. / Д. Россо, Р. Ландер, А. Бранд, Д. Харрис. — СПб.: БХВ-Петербург, 2022. — 496 с.: ил.

ISBN 978-5-9775-1210-7

Книга посвящена практическому применению платформы Kubernetes. Подробно рассматривается архитектура Kubernetes и ее составные компоненты. Описаны модели развертывания инфраструктуры, ее топология, принципы автоматизации процессов, среда выполнения контейнеров, хранилища данных и сетевое взаимодействие между элементами системы. Рассматриваются создание и маршрутизация сервисов, управление конфиденциальными данными, допусками, мультитенантность, уровни изоляции и абстрагирование. Приведены наглядные примеры развертывания Kubernetes и оркестрации контейнеров для решения различных практических задач.

Для системных архитекторов, разработчиков ПО, специалистов по информационной безопасности

УДК 004.273
ББК 32.973-018.2

Научный редактор:

Архитектор решений, руководитель группы архитекторов и системных инженеров Croc Code — *Дмитрий Бардин*

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Перевод с английского	*Михаила Райтмана*
Компьютерная верстка	*Натальи Смирновой*
Оформление обложки	*Зои Канторович*

© 2022 BHV
Authorized Russian translation of the English edition of ***Production Kubernetes***
ISBN 9781492092308 © 2021 Josh Rosso, Rich Lander, Alexander Brand, and John Harris.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Авторизованный перевод с английского языка на русский издания ***Production Kubernetes***
ISBN 9781492092308 © 2021 Josh Rosso, Rich Lander, Alexander Brand, John Harris.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc.

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-1-492-09230-8 (англ.)
ISBN 978-5-9775-1210-7 (рус.)

© Josh Rosso, Rich Lander, Alexander Brand, John Harris, 2021
© Перевод на русский язык, оформление. ООО "БХВ-Петербург", ООО "БХВ", 2022

Оглавление

ГЛАВА 1

Путь к эксплуатации

За годы своего существования платформу Kubernetes внедрили многие организации. Росту ее популярности, бесспорно, способствует распространение контейнерных рабочих заданий и наличие микросервисов. Когда команды, занимающиеся эксплуатацией, обслуживанием инфраструктуры и разработкой, сталкиваются с необходимостью создания, выполнения и поддержки этих рабочих процессов, некоторые из них выбирают в качестве решения Kubernetes. Это довольно молодой проект в сравнении с другими продуктами с открытым исходным кодом, такими как Linux. Как могут подтвердить многие клиенты, с которыми мы сотрудничали, большинство пользователей Kubernetes только начинают работать с этой платформой. Несмотря на ее применение во многих организациях, она редко встречается в эксплуатации и еще реже — в высоконагруженных проектах. В этой главе мы подготовимся к пути, на котором находятся многие команды инженеров, работающие с Kubernetes. В частности, мы рассмотрим некоторые аспекты, на которые следует обращать внимание при планировании процесса подготовки проекта к эксплуатации.

Что такое Kubernetes

Что такое Kubernetes? Платформа? Инфраструктура? Приложение? На этот счет есть много мнений, приверженцы которых могут дать вам свое определение. Вместо того чтобы высказать еще одно мнение, мы сосредоточимся на классификации тех задач, которые решает Kubernetes. После этого мы поговорим о том, как, опираясь на этот набор возможностей, достигнуть результатов, соответствующих производственному уровню. Идеальный исход прочтения этой книги — ситуация, при которой рабочие задания успешно работают с реальным трафиком.

Название *Kubernetes* в какой-то степени носит общий характер. Если заглянуть на GitHub, можно заметить, что организация kubernetes содержит (на момент написания этих строк) 69 репозиториев. А ведь есть еще и kubernetes-sigs со 107 репозиториями. Не говоря уже о сотнях проектов в составе фонда CNCF (Cloud Native Computing Foundation), существующих в этой среде! В данной книге под Kubernetes имеется в виду лишь основной проект с одноименным названием. Какой из них основной? Тот, который находится в репозитории kubernetes/kubernetes (**https://github.com/kubernetes/kubernetes**). Именно там хранятся все компоненты, которые встречаются в большинстве кластеров. От кластера, состоящего из этих компонентов, можно ожидать следующих возможностей:

- планирование распределения рабочей нагрузки для множества хостов;
- предоставление декларативного, расширяемого API-интерфейса для взаимодействия с системой;

- наличие утилиты командной строки `kubectl` для взаимодействия с API-сервером в ручном режиме;
- приведение объектов от текущего состояния к желаемому;
- предоставление базового служебного слоя для направления запросов к приложениям и обратно;
- наличие нескольких интерфейсов для поддержки подключаемых модулей, отвечающих за работу с сетью, хранилищем и т. д.

Перечисленные возможности делают данный проект, согласно утверждению его разработчиков, *системой оркестрации контейнеров, готовой к эксплуатации*. Говоря простым языком, Kubernetes дает нам возможность исполнять и распределять контейниризированные приложения на разных хостах. Помните об этом по мере того, как мы будем углубляться в подробности. Надеемся, со временем нам удастся продемонстрировать, что эта возможность, несмотря на ее основополагающий характер, является лишь одним из этапов на пути к эксплуатации.

Основные компоненты

Какие компоненты обеспечивают функциональность, которая рассматривается в этой книге? Как уже упоминалось, основные компоненты находятся в репозитории kubernetes/kubernetes. Многие из нас применяют их по-разному. Например, те, кто пользуются управляемыми сервисами наподобие Google Kubernetes Engine (GKE), скорее всего, уже имеют все эти компоненты на своих хостах. Кто-то может скачивать исходный код из репозиториев или получать подписанные версии от поставщика. Как бы то ни было, выпуски Kubernetes доступны для свободного скачивания в репозитории kubernetes/kubernetes. Распаковав такой выпуск, вы можете получить соответствующие двоичные файлы с помощью команды `cluster/get-kubebinaries.sh`. Она автоматически определит вашу целевую архитектуру и скачает серверные и клиентские компоненты. Давайте рассмотрим это на примере следующего кода и затем проанализируем ключевые компоненты:

```
$ ./cluster/get-kube-binaries.sh

Kubernetes release: v1.18.6
Server: linux/amd64  (to override, set KUBERNETES_SERVER_ARCH)
Client: linux/amd64  (autodetected)

Will download kubernetes-server-linux-amd64.tar.gz from https://dl.k8s.io/v1.18.6
Will download and extract kubernetes-client-linux-amd64.tar.gz
Is this ok? [Y]/n
```

Внутри архива со скачанными серверными компонентами, который, скорее всего, был сохранен в папку server/kubernetes-server-${ARCH}.tar.gz, находятся ключевые элементы, составляющие кластер Kubernetes:

- *API-сервер* — главное место взаимодействия для всех компонентов Kubernetes и пользователей. Именно здесь мы получаем, добавляем, удаляем и изменяем объекты. API-сервер делегирует работу с состоянием внутреннему компоненту, роль которого чаще всего играет etcd.

- *Kubelet* — агент, размещенный на хосте и взаимодействующий с API-сервером для получения информации о состоянии узла и определения того, выполнение каких приложений следует на нем планировать. В пределах хоста он взаимодействует со средой выполнения контейнеров, такой как Docker, обеспечивая корректные запуск и выполнение приложений, запланированных на соответствующем узле.
- *Диспетчер контроллеров* — группа обработчиков, которая объединена в одно приложение и занимается приведением многих важных объектов Kubernetes к желаемому состоянию. Когда такое состояние (например, "в Развертывании должно быть три реплики") объявляется, для его достижения внутренний контроллер берет на себя создание новых подов.
- *Планировщик* определяет, где должны выполняться приложения, в зависимости от того, какой узел он считает оптимальным. Для принятия этого решения он использует фильтрацию и систему оценок.
- *Kube-proxy* реализует сервисы Kubernetes, предоставляя виртуальные IP-адреса, способные направлять трафик к подам. Это достигается за счет механизма фильтрации пакетов на хосте, такого как `iptables` или `ipvs`.

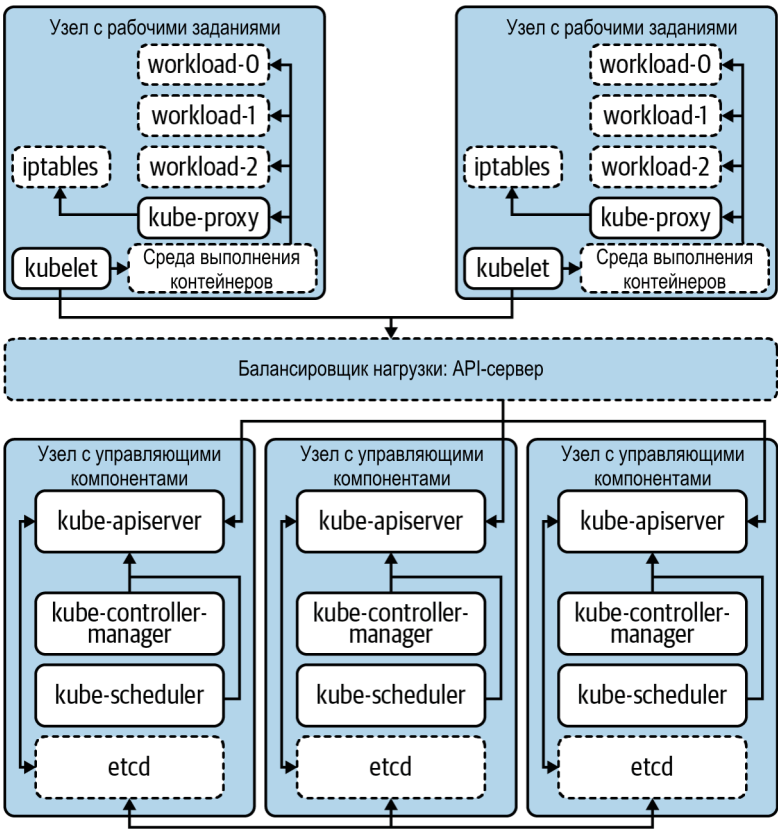

Рис. 1.1. Ключевые компоненты, составляющие кластер Kubernetes. Компоненты, обрамленные пунктирной линией, не входят в ядро Kubernetes

Несмотря на то, что это не исчерпывающий список, но в нем перечислены главные компоненты, составляющие основу той функциональности, которую мы обсудили. Если говорить в контексте архитектуры, то на рис. 1.1 показано, как взаимодействуют эти компоненты.

Архитектура Kubernetes имеет много вариантов. Например, во многих кластерах компоненты kube-apiserver, kube-scheduler и kube-controller-manager выполняются в виде контейнеров. Это означает, что мастер-узел или управляющий слой (control-plane) может также включать container-runtime, kubelet и kube-proxy. Подобные аспекты развертывания будут рассмотрены в следующей главе.

Не только оркестрация: дополнительные функции

В некоторых областях Kubernetes не ограничивается одной лишь оркестрацией приложений. Как уже упоминалось, компонент kube-proxy делает так, чтобы хосты автоматически предоставляли приложениям виртуальные IP-адреса, которые ведут к одной или нескольким внутренним подам. Очевидно, что это выходит за рамки выполнения и планирования контейнеризированных приложений. Вместо реализации всего этого в ядре Kubernetes теоретически можно было бы определить API-интерфейс для работы с сервисами, требуя, чтобы он был реализован в виде подключаемого модуля. В результате такого подхода мы были бы вынуждены выбирать между многочисленными модулями, доступными в экосистеме, вместо того чтобы задействовать функциональность, встроенную в ядро.

Это модель, которая применяется во многих API-интерфейсах Kubernetes, таких, как NetworkPolicy. Например, создание объекта `Ingress` в кластере Kubernetes не гарантирует выполнение действия. Иными словами, API-интерфейс хоть и существует, но находится за пределами ядра. Разработчики должны выбрать технологию, на основе которой этот API-интерфейс будет реализован. В случае с `Ingress` многие используют такой контроллер, как ingress-nginx (**https://kubernetes.github.io/ingress-nginx**), который работает в кластере. Он реализует API-интерфейс, считывая объекты `Ingress` и создавая конфигурационные файлы для экземпляров NGINX, которые указывают поды. Однако ingress-nginx — это лишь один из множества вариантов. Project Contour (**https://projectcontour.io**) реализует тот же Ingress API, но при этом конфигурирует экземпляры Envoy, прокси-сервера, лежащего в основе Contour. Благодаря наличию подключаемых модулей разработчикам есть из чего выбрать.

Интерфейсы Kubernetes

Развивая эту идею расширения функциональности, мы должны теперь рассмотреть интерфейсы. Интерфейсы Kubernetes позволяют нам изменять и дополнять основные возможности этой платформы. Мы относимся к интерфейсу как к определению или контракту о том, как должно происходить взаимодействие с тем или иным компонентом. В сфере разработки программного обеспечения это созвучно принципу определения функциональности, которую могут реализовывать классы или структуры. В таких системах, как Kubernetes, для удовлетворения требований этих

интерфейсов мы развертываем подключаемые модули, предоставляя возможности наподобие управления сетью.

Конкретным примером отношений между интерфейсами и подключаемыми модулями является CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров; **https://github.com/kubernetes/cri-api**). На ранних этапах развития платформа Kubernetes поддерживала всего одну среду выполнения контейнеров, Docker. И хотя Docker по-прежнему присутствует во многих кластерах, наблюдается растущий интерес к альтернативам, таким как containerd (**https://containerd.io**) и CRI-O (**https://github.com/cri-o/cri-o**). На рис. 1.2 вышеупомянутые отношения проиллюстрированы на примере этих двух сред выполнения контейнеров.

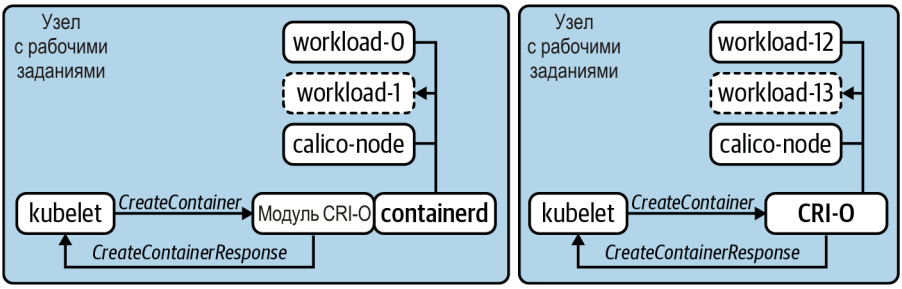

Рис. 1.2. Две разные среды выполнения контейнеров, установленных на двух узлах. Kubelet шлет такие запросы, как `CreateContainer`, определенные в CRI, в расчете на то, что среда выполнения их удовлетворит и вернет ответ

Во многих интерфейсах такие команды, как `CreateContainerRequest` или `PortForwardRequest` выполняются в виде удаленного вызова процедур (англ. Remote Procedure Calls или RPC). В случае с CRI взаимодействие происходит по gRPC, а kubelet ожидает получения ответов вроде `CreateContainerResponse` и `PortForwardResponse`. На рис. 1.2 можно также увидеть две разные модели удовлетворения требований CRI. Модуль CRI-O с самого начала создавался в качестве реализации CRI, поэтому kubelet передает ему эти команды напрямую. А вот containerd поддерживает подключаемый модуль, который служит прослойкой между kubelet и собственными интерфейсами. Какой бы ни была конкретная архитектура, главное, чтобы агенту kubelet не нужно было знать о том, как организована работа среды выполнения контейнеров в *каждом* отдельном случае. Именно этот принцип делает интерфейсы настолько мощным средством проектирования, разработки и развертывания кластеров Kubernetes.

Мы видели, как со временем часть функций была вынесена из основного проекта в соответствии с упомянутым принципом подключаемых модулей. Речь идет о вещах, которые существовали в кодовой базе kubernetes/kubernetes по историческим причинам. Примером этого являются средства интеграции cloud-provider (**https://github.com/kubernetes/cloud-provider**; CPI). Большинство модулей CPI традиционно встраивались в такие компоненты, как kube-controller-manager и kubelet. Они, как правило, отвечали за выделение балансировщиков нагрузки, предоставление доступа к метаданным облачных провайдеров и пр. Иногда, особенно

до создания CSI (Container Storage Interface — интерфейс хранилищ для контейнеров; **https://kubernetes-csi.github.io/docs/introduction.html**), эти провайдеры выделяли блочные хранилища и делали их доступными для приложений, выполняемых в Kubernetes. Это довольно обширная функциональность даже для ядра, не говоря уже о том, что ее нужно было заново реализовывать для каждого отдельно взятого провайдера! В качестве более разумного решения эти возможности были вынесены в отдельную модель интерфейса, kubernetes/cloud-provider (**https://github.com/kubernetes/cloud-provider**), реализацией которой могут заниматься разные проекты и поставщики. В результате удалось не только упорядочить кодовую базу Kubernetes, но и обеспечить возможность управления этой функциональностью вне основных кластеров Kubernetes. Сказанное относится и к таким распространенным процедурам, как обновление и устранение уязвимостей.

На сегодня существует несколько интерфейсов, которые позволяют видоизменять и дополнять возможности Kubernetes. Вот общий список, который мы расширим в следующих главах этой книги:

- CNI (Container Networking Interface — интерфейс управления сетью контейнеров) позволяет сетевым провайдерам определять, как они выполняют свои функции, от IPAM до непосредственной маршрутизации пакетов.
- CSI (Container Storage Interface — интерфейс хранилищ для контейнеров) позволяет провайдерам хранилищ удовлетворять запросы рабочих заданий в пределах кластера. Обычно реализуется для таких технологий, как vSAN и EBS.
- CRI (Container Runtime Interface — интерфейс среды выполнения контейнеров) обеспечивает поддержку различных сред выполнения, в том числе таких популярных, как Docker, containerd и CRI-O. Также способствовал распространению менее традиционных сред выполнения, включая firecracker, которая использует KVM для создания минимальных виртуальных машин.
- SMI (Service Mesh Interface — интерфейс mesh-сети сервисов) — это один из новых интерфейсов, появившихся в экосистеме Kubernetes. Он создавался в надежде унифицировать определение таких понятий, как "политика маршрутизации трафика", "телеметрия" и "управление".
- CPI (Cloud Provider Interface — интерфейс облачных провайдеров) позволяет провайдерам, таким как VMware, AWS, Azure и др., создавать для своих облачных сервисов точки интеграции с кластерами Kubernetes.
- Спецификация среды выполнения OCI (Open Container Initiative) стандартизирует форматы образов так, чтобы образы контейнеров, созданные в соответствии с ней, могли выполняться в любой среде, совместимой с OCI. Она не имеет прямого отношения к Kubernetes, но поддерживает стремление к реализации подключаемых сред выполнения контейнеров (CRI).

Kubernetes в целом

Теперь давайте поговорим о сфере применения Kubernetes. Это *средство оркестрации контейнеров* с некоторыми дополнительными функциями. Его можно рас-

ширять и изменять за счет подключаемых модулей для интерфейсов. Kubernetes может стать основополагающим инструментом для организаций, которым нужен изящный способ выполнения своих приложений. Но давайте сделаем шаг назад. Если взять системы для выполнения приложений, которые используются в вашей организации в настоящий момент, и заменить их платформой Kubernetes, будет ли этого достаточно? Во многих случаях компоненты и механизмы, составляющие "платформу приложений", имеют множество нюансов.

Нам неоднократно приходилось наблюдать за муками, вызванными необдуманным выбором так называемой "стратегии Kubernetes", когда руководство организации верит в то, что Kubernetes послужит толчком к модернизации процессов разработки и функционирования ПО. Kubernetes — это замечательная технология, но она не должна определять направление развития вашей организации в отношении инфраструктуры, платформы и/или программного обеспечения. Мы извиняемся, если вам это и так понятно, но вы бы удивились, узнав, сколько исполнительных специалистов и высокопоставленных руководителей, с которыми мы разговаривали, считают, что платформа Kubernetes сама по себе является решением проблем, в то время как на самом деле их проблемы были связаны с доставкой приложений, разработкой программного обеспечения или организационными/кадровыми вопросами. К Kubernetes лучше относиться как к одному из элементов инфраструктуры, который позволяет создавать инфраструктурную платформу для ваших приложений. Мы все ходим вокруг да около идеи о платформе приложений, поэтому давайте поговорим о том, что это такое.

Что такое платформа приложений

Одним из ключевых этапов на нашем пути к эксплуатации должно стать обсуждение идеи о платформе приложений. В нашем понимании *платформа приложений* — это подходящее место для выполнения приложений. Как и с большинством определений в данной книге, его практическое воплощение зависит от конкретной организации. Разные структурные подразделения организации ориентируются на такие желаемые результаты, как удовлетворенность разработчиков, снижение эксплуатационных расходов, сокращение времени цикла обратной связи при доставке ПО и т. д. Платформа приложений зачастую находится на пересечении приложений и инфраструктуры. Удобство разработки (англ. developer experience или devx) обычно является ключевым принципом в этой области.

Платформы приложений бывают разными. Некоторые в основном инкапсулируют такие основополагающие компоненты, как IaaS (например, AWS) или средство оркестрации (например, Kubernetes). Отличным примером этой модели является Heroku. С помощью данного сервиса вы можете взять проект, написанный на таких языках, как Java, PHP или Go, и развернуть его в реальных условиях эксплуатации с помощью одной команды. Ваше приложение работает в окружении множества сервисов, которые пришлось администрировать самостоятельно, если бы они не предоставлялись самой платформой. Имеются в виду такие вещи, как средства сбора метрик, сервисы для работы с данными и непрерывная доставка (англ. Continuous

Delivery или CD). Платформа также дает вам базовые конфигурации для запуска набора наиболее распространенных приложений, которые могут легко масштабироваться. Используется ли Kubernetes в Heroku? Есть ли у Heroku собственные центры обработки данных (ЦОД), или все работает поверх AWS? Какая разница? Для пользователей Heroku эти детали неважны. Важно то, что эти задачи делегируются провайдеру или платформе, что позволяет разработчикам больше времени уделять решению бизнес-проблем. Такой подход применяется не только в облачных сервисах. В OpenShift от RedHat принята похожая модель, в которой Kubernetes является, скорее, одной из деталей реализации, а разработчики и администраторы платформы взаимодействуют с набором более высокоуровневых абстракций.

Почему бы на этом не остановиться? Если такие платформы, как Cloud Foundry, OpenShift и Heroku уже решили эти проблемы за нас, зачем возиться с Kubernetes? Существенным недостатком многих готовых платформ приложений является то, что они навязывают нам свои "взгляды". Делегирование всех обязанностей по управлению и обслуживанию внутренней части системы позволяет избежать большого объема работ по сопровождению. Но в то же время, если методы обнаружения сервисов или управления конфиденциальными данными, которые применяет платформа, не соответствуют вашим организационным требованиям, вы можете быть лишены возможности решить эту проблему. Кроме того, есть такое понятие как зависимость от поставщика или от чужого мнения. Абстракции влекут за собой определенные взгляды на то, как должны проектироваться, упаковываться и развертываться ваши приложения. Это означает, что переход на другую систему может оказаться непростым. Например, рабочие задания намного проще перенести между Google Kubernetes Engine (GKE) и Amazon Elastic Kubernetes Engine (EKS), чем между EKS и Cloud Foundry.

Спектр подходов

Как вы уже сами видите, существует несколько подходов к созданию успешной платформы приложений. Давайте сделаем для демонстрации несколько далеко идущих предположений и сравним теоретические компромиссы, свойственные разным подходам. На рис. 1.3 за пример взята типичная компания среднего/крупного размера, с которой мы обычно имеем дело.

В левой нижней части мы видим развертывание самих кластеров Kubernetes, что требует относительно небольших усилий со стороны инженеров, особенно если за обслуживание управляющего уровня отвечает сторонний сервис вроде EKS. Такой подход обеспечивает не слишком высокую готовность к промышленному внедрению, так как большинство организаций сталкивается с необходимостью в дополнительной работе поверх Kubernetes. Но в некоторых ситуациях одной лишь платформы Kubernetes может быть достаточно, например, при использовании отдельных кластеров.

В правой нижней части находятся более зрелые платформы, которые предоставляют разработчикам полный набор готовых возможностей. Cloud Foundry — отличный пример проекта, который решает множество задач, возлагаемых на платформу

приложений. Программное обеспечение, работающее в Cloud Foundry, не должно выходить за те рамки, которые очертили разработчики этой платформы. А вот платформа OpenShift, которая куда лучше готова к промышленному использованию, чем просто Kubernetes, дает бóльшую свободу выбора относительно того, как ее настраивать. Что собой являет эта гибкость: преимущество или лишнюю работу? Это ключевой вопрос, на который вы должны ответить.

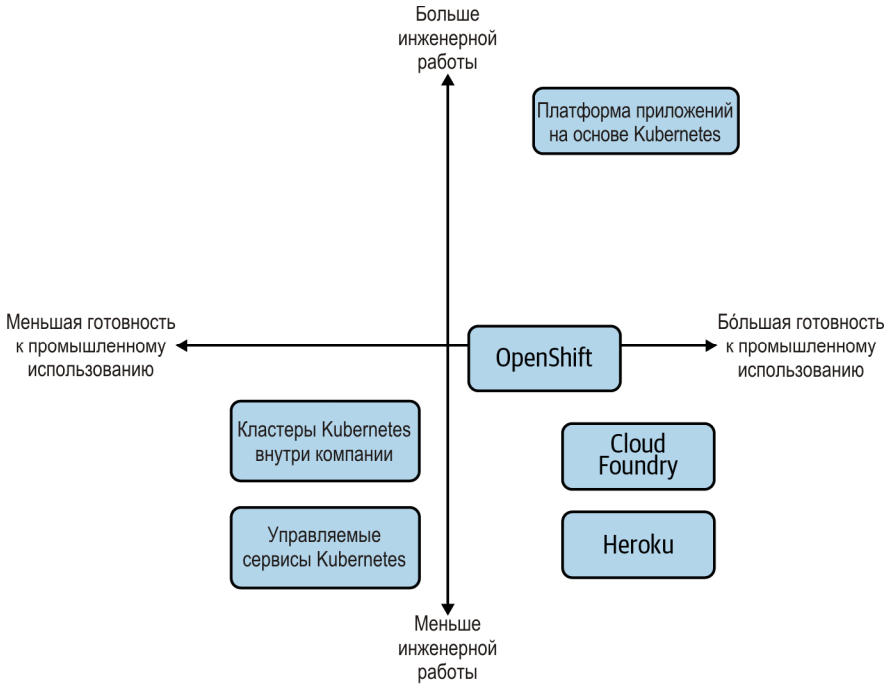

Рис. 1.3. Разнообразные способы организации платформы приложений для разработчиков

Наконец, справа вверху представлено создание платформы приложений поверх Kubernetes. Это, несомненно, требует больше всего инженерных усилий по сравнению с остальными вариантами, по крайней мере, с точки зрения платформы. Но благодаря расширяемости Kubernetes вы можете получить результат, который будет лучше соответствовать вашим потребностям в плане разработки, инфраструктуры и бизнеса.

Спектр подходов с учетом потребностей вашей организации

Диаграмме на рис. 1.3 недостает третьего измерения, оси *Z*, которая показывает, насколько тот или иной подход соответствует вашим требованиям. Рассмотрим еще одно графическое представление. На рис. 1.4 показано, как это может выглядеть, если учитывать соответствие платформы потребностям организации.

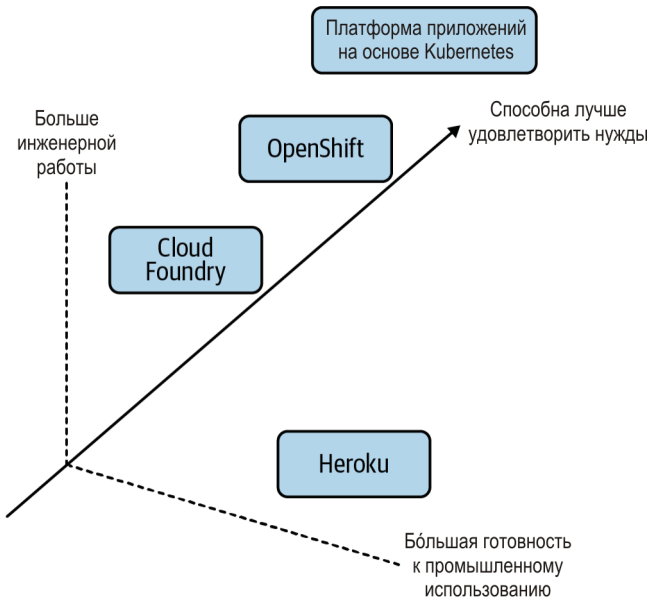

Рис. 1.4. Дополнительное измерение (ось *Z*), сопоставляющее варианты с потребностями вашей организации

С точки зрения требований, возможностей и ожидаемого поведения, самостоятельное создание платформы почти всегда дает наиболее подходящий результат. Или, по крайней мере, наиболее близкий к таковому. Ведь вы можете создать что угодно! Теоретически вы могли бы воссоздать в своей компании Heroku на основе Kubernetes, с небольшими изменениями функциональности, но при этом следует учитывать соотношение затрат и полученной выгоды (оси *X* и *Y*). Давайте добавим в наш разговор больше конкретики и рассмотрим некоторые требования к платформе приложений следующего поколения:

- регламент обязывает размещать основную часть системы локально;
- вам нужно поддерживать свои многочисленные физические серверы вместе с ЦОД, в котором используется vSphere;
- вы хотите удовлетворить растущий спрос со стороны разработчиков на упаковывание приложений в контейнеры;
- вам нужно создать API-интерфейс для механизмов самообслуживания, которые позволят отойти от выделения инфраструктурных ресурсов "на основе тикетов";
- вы хотите убедиться в том, что разрабатываемые вами API-интерфейсы не завязаны на конкретного поставщика, так как в прошлом миграция с такого рода систем стоила вам миллионы;
- вы не против платить за корпоративную поддержку разнообразных продуктов в своем стеке, но не в восторге от моделей, в которых весь стек лицензируется для каждого узла, ядра или экземпляра приложения.

Чтобы определить, является ли построение платформы приложений разумным начинанием, мы должны понимать, насколько "созрели" наши инженерные возможности, есть ли у нас желание заниматься формированием и развитием групп разработчиков, и обладаем ли мы достаточными ресурсами.

Платформы приложений: подводим итоги

Следует признать, что определение платформы приложений остается довольно размытым. Мы сосредоточились на разнообразных платформах, которые, как нам кажется, дают командам разработчиков нечто большее, чем просто оркестрацию приложений. Мы также ясно дали понять, что Kubernetes можно изменять и дополнять для получения похожих результатов. Выводя свое мышление за рамки того, как внедрить Kubernetes, и задаваясь вопросами наподобие "как сейчас организован рабочий процесс разработчиков, какие у них проблемы и желания?", группы сотрудников, которые занимаются платформой и инфраструктурой, будут более успешными в своей профессиональной деятельности. Можно утверждать, что, сосредоточившись на последнем, вы с большей вероятностью наметите себе подходящий путь к эксплуатации и справитесь с непростым процессом внедрения. В конечном счете мы хотим удовлетворить требования к инфраструктуре, безопасности и удобству разработки, чтобы наши клиенты (которыми обычно являются разработчики) получили решение, отвечающее их потребностям. Как правило, мы не стремимся предоставить "мощный" движок, поверх которого каждый разработчик должен создать собственную платформу, что с юмором продемонстрировано на рис. 1.5.

Рис. 1.5. Если разработчики хотят иметь готовое решение (например, полноценный автомобиль), не надейтесь, что им будет достаточно двигателя без кузова, колес и прочего

Создание платформ приложений на основе Kubernetes

Итак, мы воспринимаем Kubernetes как один из этапов нашего пути к развертыванию. Учитывая это, справедливым будет следующий вопрос: "Может, у Kubernetes просто недостаточно возможностей?" Проект Kubernetes был вдохновлен одним из важнейших принципов философии Unix: "пишите программы, которые делают что-то одно и делают это хорошо". Мы считаем, что лучшими возможностями этой платформы являются те, которых у нее нет! Это становится еще очевидней после

мучений с универсальными платформами, которые пытаются решать за вас все проблемы человечества. Блестящая черта платформы Kubernetes — ее узкая направленность: она пытается быть хорошим средством оркестрации, но в то же время предоставляет четкие интерфейсы, поверх которых можно написать что-то свое. Таким образом, платформу можно сравнить с фундаментом дома.

Хороший фундамент должен быть прочным, подходить для возведения остальных конструкций и предоставлять подходящие "интерфейсы" для проведения коммуникаций. Несмотря на важную роль, сам по себе фундамент редко когда становится подходящим местом для проживания. Обычно на фундаменте должен находиться какой-то дом. Прежде чем обсуждать *строительство* поверх такого фундамента, как Kubernetes, давайте рассмотрим готовое меблированное жилище, показанное на рис. 1.6.

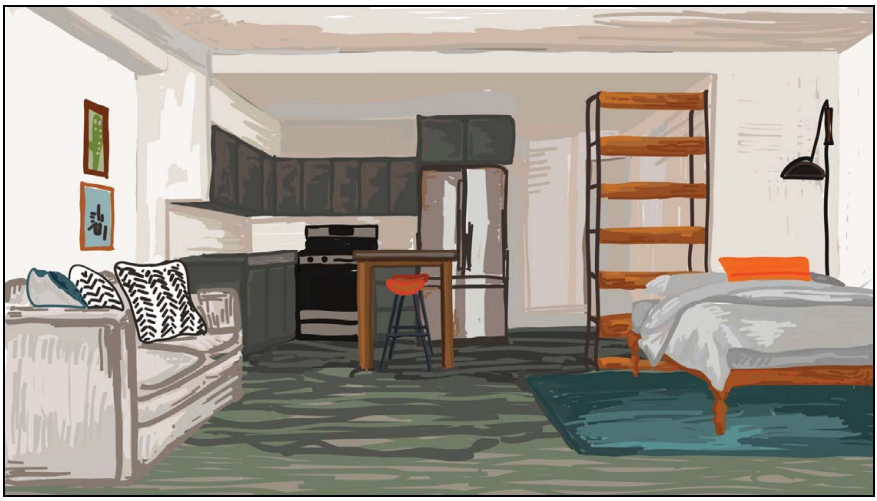

Рис. 1.6. Апартаменты, готовые к заселению. Похожи на PaaS, такие как Heroku (иллюстрация Джессики Аппельбаум)

Этот вариант жилья, подобный нашим примерам с Heroku, подходит для проживания и не требует дополнительных вложений. Вы, конечно, можете немного изменить интерьер, но большинство проблем уже решено. Если подходит арендная плата, и вы согласны смириться с тем, как все устроено внутри, вы можете жить припеваючи с самого первого дня.

Но вернемся к платформе Kubernetes, которую мы сравнили с фундаментом. Поверх нее можно построить жилой дом, как показано на рис. 1.7.

Инвестируя в планирование, проектирование и поддержку, мы можем создавать замечательные платформы, способные выполнять рабочие задания в разных организациях. Это означает, что мы контролируем каждый аспект конечного результата. Дом может и должен быть приспособлен к нуждам его будущих жильцов (наших приложений). Давайте теперь разберем различные слои и соображения, благодаря которым это возможно.

Рис. 1.7. Строительство дома. Похоже на создание платформы приложений, в основе которой лежит Kubernetes (иллюстрация Джессики Аппельбаум)

Начиная снизу

Мы должны начать с самого низа, в том числе с того, какие технологии будет использовать Kubernetes. Обычно речь идет о центре обработки данных или облачном провайдере, который предлагает вычислительные ресурсы, хранилище и доступ к сети. Дальше на основе этого можно развертывать Kubernetes. Всего за несколько минут вы можете получить кластер, который работает поверх внутренней инфраструктуры. Kubernetes можно развернуть несколькими способами, которые мы подробно обсудим в *главе 2*.

Следующий важный аспект существования кластеров Kubernetes, который поможет нам определить, что следует создавать поверх них, — процесс внедрения Kubernetes. Его ключевые элементы представлены на рис. 1.8.

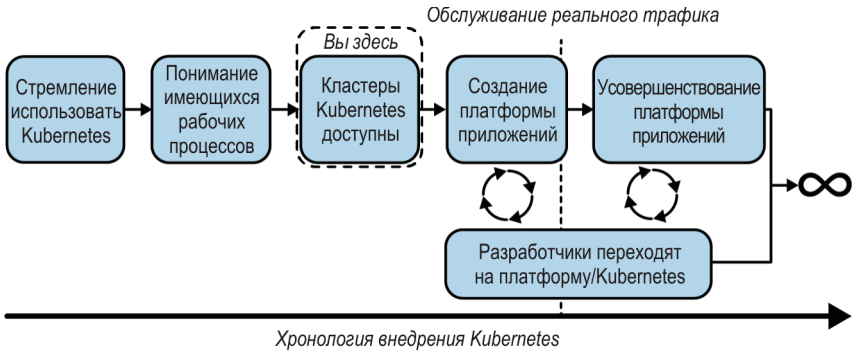

Рис. 1.8. Процесс, через который могут пройти наши команды разработчиков на пути к полноценному внедрению Kubernetes

С появлением Kubernetes в вашей организации, не удивляйтесь, если к вам начнут обращаться с подобными вопросами:

- "Как сделать так, чтобы трафик между приложениями шифровался?"
- "Как сделать так, чтобы исходящий трафик проходил через шлюз, который гарантирует CIDR?"
- "Как организовать трассировку и дашборды для приложений?"
- "Как проводить погружение разработчиков, не требуя от них глубоких знаний Kubernetes?"

Этот список можно продолжать бесконечно. Зачастую нам самим приходится определять, какие требования должны удовлетворяться на уровне платформы, а какие — на уровне приложения. Главное здесь — иметь глубокое понимание существующих рабочих процессов, чтобы создаваемая нами платформа отвечала текущим ожиданиям. Если мы не в состоянии обеспечить нужную функциональность, как это повлияет на группы разработчиков?

Дальше мы можем приниматься за создание платформы поверх Kubernetes. Очень важно, чтобы в ходе данного процесса мы действовали совместно с разработчиками, желающими опробовать нашу платформу на ранних этапах, и знали об их впечатлениях, поскольку это поможет нам принимать обоснованные решения с учетом оперативно поступающих отзывов. Такой процесс не должен прекращаться даже после развертывания платформы в промышленном окружении. Не рассчитывайте на то, что платформа получится статической, и что разработчики будут пользоваться ею на протяжении десятилетий. Чтобы достичь успеха, мы должны постоянно взаимодействовать с нашими группами разработчиков, что позволит нам узнавать о проблемах и недостающих возможностях, которые могли бы ускорить темпы разработки. Для начала будет неплохо подумать о том, какой уровень взаимодействия с Kubernetes мы должны ожидать от наших разработчиков. В итоге мы уясним, насколько общими должны быть наши абстракции.

Спектр абстрагирования

В прошлом нам встречались несколько надменные высказывания вроде "Если вашим разработчикам известно о том, что они используют Kubernetes, это провал!" Такой взгляд на взаимодействие с Kubernetes может быть вполне приемлемым, особенно если вы создаете продукты или услуги, потребителям которых все равно, какая технология оркестрации находится внутри. Например, вы можете работать над системой управления базами данных, которая поддерживает разные БД. Ваших разработчиков, скорее всего, не интересует, внутри чего выполняются сегменты или экземпляры базы данных: Kubernetes, Bosh или Mesos! Однако делать эту философию основным критерием успеха вашей команды будет опасно. По мере создания поверх Kubernetes платформы с сервисами для лучшего обслуживания наших клиентов нам неоднократно придется принимать решения о том, как должны выглядеть наши абстракции. Это проиллюстрировано на рис. 1.9.

Данный вопрос может не давать покоя тем, кто занимается созданием платформы. Абстракции приносят много пользы. Такие проекты, как Cloud Foundry, предостав-

ляют разработчикам полнофункциональное окружение, например, с помощью одной единственной команды `cf push` мы можем взять приложение, собрать его и развернуть для обслуживания реального трафика. Сосредоточившись на достижении этой цели и описанных возможностей, Cloud Foundry углубляет свою поддержку работы поверх Kubernetes, и мы ожидаем, что этот переход будет выглядеть, скорее, как аспект реализации, чем как изменение функциональности. Еще одна закономерность, которую мы наблюдаем, состоит в желании предлагать компаниям нечто большее, чем просто Kubernetes, но при этом не вынуждать разработчиков делать сознательный выбор между разными технологиями. Например, в некоторых компаниях параллельно с Kubernetes используется Mesos. Такие компании создают абстракцию, позволяющую прозрачно выбирать место размещения приложений, не перекладывая эту обязанность на разработчиков, что также позволяет им избежать зависимости от конкретной технологии. Противоположный подход — создание абстракции поверх двух систем, которые работают по-разному. Это требует от компании зрелости в принятии решений и существенных инженерных усилий. К тому же, несмотря на то, что разработчикам больше не нужно знать, как взаимодействовать с Kubernetes или Mesos, они должны уметь работать с абстрагированной системой, существующей только в их компании. В наш век открытого исходного кода разработчики, какой бы стек технологий они ни использовали, не очень-то любят изучать системы, опыт работы с которыми будет бесполезен в других организациях.

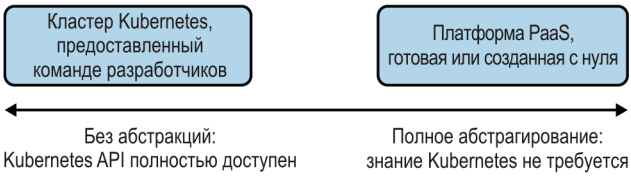

Рис. 1.9. Противоположные концы спектра: предоставление каждой команде отдельного кластера Kubernetes или полная инкапсуляция Kubernetes в виде платформы как сервиса (англ. Platform as a Service или PaaS)

Наконец, еще одна ловушка, которая нам встречалась, — "одержимость абстракциями", которая не дает предоставить доступ к ключевым возможностям Kubernetes. Со временем это может превратиться в игру в кошки-мышки, когда мы пытаемся не отставать от проекта, в результате чего наша абстракция может стать такой же сложной, как и система, которую мы абстрагируем.

На противоположном конце спектра находятся создатели платформ, которые хотят предложить разработчикам кластеры с самообслуживанием, которые тоже могут быть отличной моделью. Ответственность за взаимодействие с Kubernetes перекладывается на команды разработчиков. Понимают ли они, как работают Развертывания (Deployments), Наборы реплик (Replicas), Pod, Сервисы и API-интерфейсы объектов `Ingress`? Умеют ли они обращаться с единицами измерения вычислительных ресурсов, milliCPU, и знают ли они, как происходит выделение недоступных ресурсов (overcommit)? Известно ли им, как добиться того, чтобы выполнение приложений, сконфигурированных с более чем одной репликой, всегда планировалось на разных узлах? Если да, то это идеальная возможность избежать чрезмерного ус-