АННОТАЦИЯ

!!! Скорректировать цель и остальное согласно тому, что в итоге выйдет в ВКР

Цель выпускной квалификационной работы – исследование и создание кластерной инфраструктуры, которая позовлит разрабатывать и развертывать программные проекты множеством команд из нескольких человек.

В работе составлен обзор по архитектурам приложений и инфраструктур. Произведен анализ инструментов для создания инфраструктуры.

В ходе работы в рамках проекта был развернут кластер Kubernetes, его вспомогательные компоненты, настроено их взаимодействие.

Созданная инфраструктура внедрена в МИЭМ НИУ ВШЭ и происходит поэтапный процесс переноса проекта в нее.

В структуру работы входит введение, ??? глав, заключение, список использованных источников , 3 прил. , 29 рис. и 29 источн.

Общий объем проекта составляет 47 стр.

ABSTRACT

blablabla

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время существует спрос на реализацию решений по автоматизации процессов разработки и развертывания программных продуктов. Для этого используются такие средства, как:

— Система контроля версий;

— Средства непрерывной интеграции и доставки артефактов;

— Система контенеризации;

— Оркестрация контейнеров;

— Вспомогательные инструменты.

На текущий момент в МИЭМ активно развивается проектная деятельность. Но изначально инфраструктура не была готова к такому большому количеству проектов и команд, реализующих эти проекты. В первую очередь, необходима была платформа для удобного хранения разработок, то есть система контроля версий. После этого данные наработки необходимо было где-то развернуть. Ранее это представляло большую многоэтапную задачу, от поиска локальных администраторов и получения мощностей, до получения доменного имени для публикации проекта для промышленного использования.

Цель работы – уменьшение метрик времени вывода в тестовую и промышленную эксплуатацию программных продуктов проектных команд МИЭМ НИУ ВШЭ посредством переиспользования ресурсов контейнерных сред и общих инфраструктурных кластеров.

СТАРАЯ ТЕРМИНОЛОГИЯ

Immutable - неизменный.

Self-healing - самолечение.

Declarative - декларативность.

SLA - доступность.

Daemon или **Демон** - процесс, запущенный в фоне.

CLI (Command line interface) - консольный интерфейс приложения.

FS (File system) - файловая система раздела.

Hash или **Хэш** - результат выполнения некоторой математической хэш-функции.

Tag или **Тэг** - некоторая отметка на чем-либо для более удобного поиска объектов.

Yaml (YAML Ain't Markup Language) -"дружественный"формат сериализации данных, концептуально близкий к языкам разметки.

SSD (Solid State Drive) или *Твердотельный диск* - вид накопителя, хранящего данные.

REST API - архитектурный стиль взаимодействия компонентов распределенного приложения в сети.

Garbage Collector - "сборщик мусора".

QoS (Quality of Service) - органичения на что-либо, разграничающие по ролям.

Requested resources - запрошенные ресурсы.

Netwrok Policies - сетевые политики.

Iptables - обертка над netfilter.

Ipvs - обертка над netfilter.

OSI - сетевая модель стека сетевых протоколов.

VRRP (Virtual Router Redundancy Protocol) - сетевой протокол, предназначенный для увеличения доступности маршрутизаторов.

SSL/TLS (Secure Sockets Layer) (Transport Layer Security) - криптографические протоколы, обеспечивающие защищенную передачу данных между узлами в сети Интернет.

Hook - перехват чего-либо.

Git - одна из систем контроля версий.

HTTP (HyperText Transfer Protocol) - протокол прикладного уровня передачи данных.

Tar - архиватор.

GET Requests - метод *HTTP*.

1 Исследовательская часть работы

1.1 Виды архитектур инфраструктуры и приложений в ней

На текущий момент существует множество решений со стороны архитектуры инфраструктуры и, как следствие, приложений в ней. Для выбора конкретного решения под вышеуказанные цели необходимо рассмотреть основные из них. Далее они будут описаны по хронологии появления.

а) Монолитная эра. Ей свойственны следующие аспекты:

- Приложения монолитные;
- Большое количество зависимостей;
- Долгая разработка до релиза;
- Все инстансы известны по именам;
- Использование виртуализации. Это означает:
  - Один сервер – несколько виртуальных машин (далее – ВМ или VM) [1];
  - Resource Management;
  - Изоляция окружений.

Соответсвенно использовались VM:

- VMWare;
- Microsoft Hyper-V;
- VirtualBox;
- Qemu.

Подход был следующий: один большой сервер делили на несколько виртуальных машин. Это давало полную изоляцию, но недостатками были:

— Hypervisor [2];

— Большие образы;

— Как следствие больших образов с разным ПО – медленное управление VM.

б) На смену им пришла виртуализация на уровне ядра с помощью следующих инструментов:

— OpenVZ;

— Systemd-nspawn;

— LXC.

Но остались прежние проблемы:

— Большие образы с операционной системой (далее – ОС или OS) с большим количеством ПО;

— Нет стандарта упаковки и доставки;

— DLL Hell [3].

в) Но далее пришли контейнеры. Разница между VM и контейнером:

— Виртуальная машина подразумевает виртуализацию железа для запуска гостевой ОС;

— Контейнер использует ядро хостовой ОС;

— В VM может работать любая ОС;

— В контейнере может работать только GNU/Linux (с недавних пор и Windows);

— VM хороша для изоляции;

— Контейнеры не подходят для изоляции.

В итоге мы приходим к ситуации, изображенной далее (рис. 1):

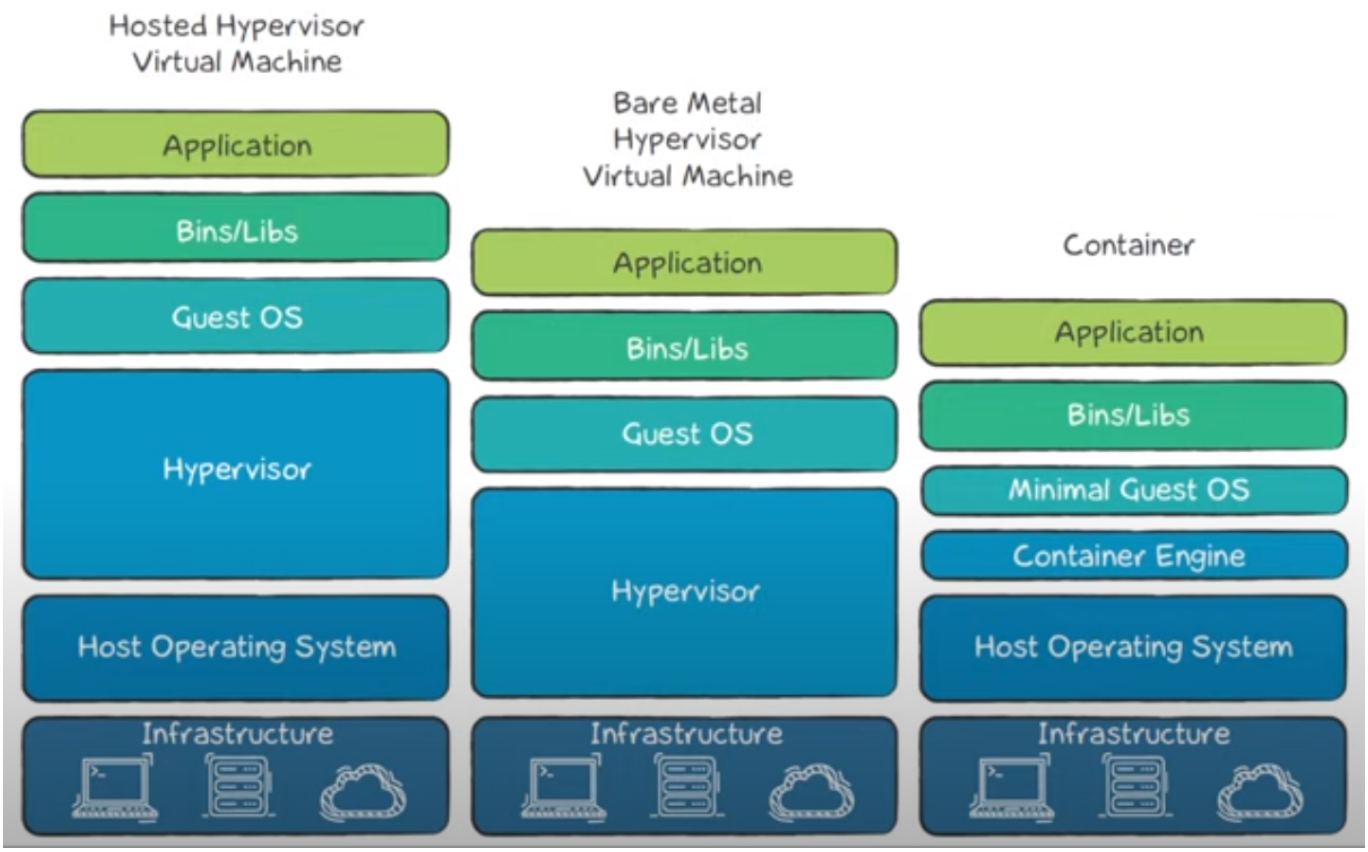

Рис. 1. Сравнение VM и контейнеров.

Способ реализации контейнеризации:

— Namespaces:
  — Process ID (далее – PID);
  — Networking;
  — Mount;
  — User.
— Control Groups (далее – Cgroup):
  — Memory;
  — CPU;
  — Block Input/Output (далее – I/O);
  — Network.

В итоге получается следующая логика:

— Один процесс – один контейнер;

— Все зависимости в контейнере;

— Чем меньше образ контейнера (файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера) [4] – тем лучше;

— Инстансы становятся эфемерными.

И в 2014-2015 годах Docker [5] приобрел популярность за счет следующих аспектов:

— Меняет философию подхода к разработке;

— Стандартизует упаковку приложения;

— Решает вопрос зависимостей;

— Гарантирует воспроизводимость;

— Обеспечивает минимум дополнительных средств для использования.

!!! МБ ДОБАВИТЬ ПРО CONTAINERD

На основе этих данных можно сделать вывод, что использование контейнеризации и, в частности, Docker позволит сущетсвенно сократить метрики по прохождению этапов от разработки кода до промышленного использования при использовании микросервисной архитектуры.

1.2 Область применения

В первую очередь данная работа ориентирована на потребности проектной деятельности МИЭМ НИУ ВШЭ. Но данное решение применимо в любой организации, где разрабатывается большое количество микросервисных приложений разными командами.

1.3 Обзор литературы

В первую очередь интересны и полезны статьи от IBM [6][7]. Начинаются они со сравнения виртуальных машин и контейнеров, которое было приведено выше в разделе 1.1 видов архитектуры. Затем идет перечисление и детальное рассмотрение каждого из преимуществ использования контейнеров в архитектуре инфраструктуры.

Статья [8] освещает сравнение двух популярнейших систем Continuous Integration и Continuous Delivery или Deployment (далее – CI/CD, этапы непрерывной интеграции разработок и непрерывной доставки кода вплоть до промышленного использования) [9] – Jenkins и GitLab CI/CD. В ней рассмотрены основные особенности, функциональные возможности и преимущества двух систем. Затем дается достаточно объективное сравнение этих инструментов, что позволяет выбрать необходимый под создаваемую инфраструктуру.

По аналогии с сравнением систем CI/CD, стоит обратить внимание на сравнение систем оркестрации контейнеров [10]. Среди рассматриваемых инструментов: Swarm, Kubernetes, Mesos, Cattle. Как один из немногих недостатков Kubernetes можно отметить отсутствие ограничений на обращение к диску. Но в тот же момент, среди рассматриваемых систем этим преимуществом обладает только Mesos. В остальном же Kubernetes выигрывает по функциональности с большим отрывом. Если посмотреть на приведенные в публикации графики, то можно увидеть, что есть не один случай, когда Kubernetes выигрывает с большим отрывом от других средств. В других же случаях он на равных с большинством.

Если рассмотреть прикладные статьи по тематике контейнеризации, то внимания заслуживает следующий материал [11]. В нем четко сформулированы простые правила написания инструкций для сборки Docker Images. Как аналогия приводится написание кода на языке программирования C или C++. Также полезны листинги с примерами файлов-инструкций.

В дополнение к прикладным статьям, стоит обратить внимание на еще одну [12]. В ней также рассмотрены преимущества контейнеризации, ее особенности. Из ключевых стоит отметить:

— Простоту;

— Поддерживаемость;

— Устойчивость;

— Воспроизводимость;

— Удобство;

— Размер;

— Прозрачность.

Нельзя обойти стороной рассмотрение официальных лучших практик по контейнеризации от разработчиков самого Docker [13]. Первым и, пожалуй, ключевым они выделяют важность порядка инструкций, потому что если что-то изменяется выше, то все последующие инструкции будут тоже исполнятся заново, что сильно увеличивает время сборки Docker Image.

В другой статье [14] приводится сравнение вертикального и горизонтального масштабирования. Это давняя известная проблема: вместо наращивания мощности одного инстанса, например, сервера, лучше взять еще один такой же конфигурации. Это позволяет экономить финансы. Но опять же, приложение архитектурно должно быть к этому готово.

Если же рассмотреть промышленное использование в больших масштабах, то стоит обратить внимание на данную книгу [15]. Она рассказывает о «кровавом энтерпрайзе», что позволяет выявить ряд недостатков в выбранной архитектуре, так как не всегда она подходит под нужды бизнеса.

Для сравнения систем оркестрации можно изучить статью об одной из них и ее использовании [16]. С одной стороны, она тоже намного мощнее такого инструмента как docker-compose, который подходит для одного

микросервиса. Но в сравнении с Kubernetes есть ряд очень ощутимых преимуществ у последнего.

Чтобы лучше понять логику «пакетного менеджера для приложений» – Helm, стоит обратить внимание на статью о его истории и будущем [17]. Как минимум, заслуживает внимание разбор гигантское обновление со второй на третью версию этого инструмента, где отказались от части логики, из-за чего отсутствует обратная совместимость.

Для рассмотрения связки GitLab и Kubernetes отлично подходит статья компании, которая помогает внедрять эти технологии огромному количеству малого и среднего бизнесов – Flant [9]. Более того, эта статья заслуживает внимания, так как является выдержкой с выступления на одной из крупнейших конференций в России по высоконагруженным системам – Highload++ 2017. В первую очередь хочется отметить наглядность анимированного изображения, которое позволяет наглядно увидеть разницу между разными инструментами CI/CD. Данное изображение дает понять, что:

— Git (одна из систем контроля версий) [18] вместе с shell дает несколько окружений и анализ кода;

— Добавление Docker позволит улучшить тестирование до тестов без окружения и добавит аспект Stateless приложения в архитектуру;

— Дополнительное использование Kubernetes и Helm позволяет довести тестирование до тестов в «полном» окружении, а архитектуру привести к микросервисной логике;

— При пополнении стека с помощью GitLab мы получаем несколько площадок вместо нескольких окружений, а также простое разделение прав доступа. Этого набора уже зачастую предостаточно для покрытия нужд;

— В заключение – GitLab Enterprise, который привносит разные права на окружения, «multi stage approval» или же «quorum approval» логики.

В итоге у компании Flant используется один из самых популярных стеков технологий, за исключением dapp, который на текущий момент переименован в werf, их личная разработка, которая используется для упрощения или улучшения процессов сборки и не только.

В материалах предыдущей статьи есть ссылки на другие публикации, что позволяет глубже погрузиться в тематику. Если посмотреть на обзор другого доклада с RootConf 2017 [19], то можно заметить пересечения с вышеупомянутой историей развития архитектуры. При этом освещены вопросы, которые не обсуждались ранее, а именно ряд сложностей микросервисной архитектуры:

— Сбор логов;

— Сбор метрик;

— Проверка состояния сервисов и их перезапуск в случае проблем;

— Автоматическое обнаружение сервисов;

— Автоматизация обновления конфигураций компонентов инфраструктуры (при добавлении/удалении новых сущностей сервисов);

— Масштабирование;

— CI/CD;

— Зависимость от выбранного «поставщика решения».

По мнению докладчика, все эти сложности можно закрыть с помощью Kubernetes и ряда вспомогательных инструментов.

Далее в статье [19] освещаются базовые аспекты архитектуры Kubernetes, его сущности от Pod до Ingress. Следующий раздел этой публикации погружает нас в разные архитектуры Kubernetes, в зависимости от нагрузки, требований к отказоустойчивости и других параметров, что еще раз демонстрирует его гибкость как инструмента.

Если изучить еще одну статью [20], которая затрагивает первые практики Continuous Delivery с Docker, то можно раскрыть детальнее первый из этих терминов. Под Continuous Delivery имеется в виду последовательность действий, в результате который код из Git-репозитория сначала собирается, потом тестируется, после чего попадает в промышленное использование и после уходит в архив. Также в статье поднимается вопрос проблемы простоя во время выкатки в промышленное использование новой версии продукта. На текущий момент это решается следующей последовательностью:

— Старая версия запущена;

— В соседнем месте запускается и “прогревается” новая версия;

— Когда новая версия готова к работе, трафик переключается на нее;

— Старая версия может быть остановлена.

Не меньшего внимания заслуживает еще одна статья от компании-интегратора Flant уже про базы данных в Kubernetes [21]. Основная проблематика заключается в том, что Kubernetes ориентирован на stateless приложения, то есть приложения без сохранения состояния. А БД – яркий пример stateful приложения, которому жизненно необходимо сохранение своего состояния. В старой архитектуре приложений была следующая логика с СУБД: репликация на двух железных серверах с резервированным питанием диском сетью и всем остальным, включая инженера на дежурной смене. Это позволяло гарантировать, что если что-то или кто-то выйдет из строя, то есть возможность оперативно переключиться или заменить неисправный элемент. В архитектуре же Kubernetes ощутимо другая логика, но она тоже привносит отказоустойчивость:

— Логика кворума вместе с логикой логикой активного и запасного мастер-компонентов, которые управляют кластером;

— Автоматический «переезд» сущностей с упавшего сервера на остальные;

— Возможность декларативно описать логику поведения при недоступности сущности.

Как простое решение с низким показателем критичности в аспекте отказоустойчивости предлагается сущность StatefulSet с одной сущностью Pod, что означает запуск СУБД или другого stateful приложения в одном экземпляре. Этого может быть вполне достаточно для тестовых сред. Чуть более сложная схема – StatefulSet уже с двумя инстансами, между которыми настроена репликация данных с возможностью переключение с активного приложения на запасное. Но оптимальным решением будет рассмотреть приложения с сохранением состояния, которые уже умеют работать в кластере, например Kafka в роли брокера сообщений.

В одной из перечисленных выше статей поднималась проблема мониторинга большого количества сущностей в архитектуре Kubernetes. Есть статья [22], которая освещает именно эти моменты. Это также обзор с доклада, в этом случае с RootConf 2018. В его начале рассказывается о ключевых аспектах мониторинга, например, что спидометр показывает скорость и усреднение его редких показателей будут сильно расходится с одометром. Также освещены специфики мониторинга в Kubernetes, основной из которых является совершенно другой масштаб того, что нужно мониторить и с какой скоростью. В статье утверждается, что ключевой выбор для такого мониторинга – Prometheus. Далее уже речь заходит об архитектуре самого Prometheus, и о том, какие метрики стоит собирать в Kubernetes. Для наглядного отображения всех агрегированных метрик используется такой инструмент как Grafana.

Кроме вышеперечисленных статей, начинающему стоит обратить внимание на пример или инструкцию развертывания простого проекта в Kubernetes с помощью GitLab CI/CD [23]. Если предварительно посетить и изучить курсы по GitLab, Docker, Kubernetes и Helm, то данный материал будет предельно понятен и поможет начать создание шаблонов для развертывания продуктов в инфраструктуре.

Еще одна проблема – накопление и, как следствие, необходимость умной очистки неиспользуемых Docker Images, освещается в статье от вышеупомянутой компании Flant [24]. Есть два решения данной проблемы: либо использовать фиксированное количество тегов для Docker Images, либо же каким-то образом очищать Docker Images. Во втором случае необходимо выбрать критерии актуальности образа, что также освещено в статье. Ключевой замысел реализован в подборе оптимального количества последних сохраняемых образов для каждой ветки Git-репозитория.

Выводы по рассмотренным источникам приведены в разделе 1.4.

1.4 Разработка архитектуры и выбор инструментария

На основе всего вышеописанного можно сделать вывод, что на текущий момент среди архитектур программных проектов лидирует микросервисная архитектура, за исключением ситуаций, когда она неприменима.

Среди систем контроля версий лидером является Git. А самой востребованной реализацией сервера Git, которую можно расположить на собственных вычислительных мощностях, является GitLab.

В микросервисной архитектуре для удобства упаковки, доставки и развертывания продукта используются контейнеры. Лидер среди них – Docker.

Для того чтобы расположить контейнеры Docker на нескольких серверах, удобно их администрировать, масштабировать, получать с них метрики для мониторинга, а также других связанных задач, используется оркетстратор контейнеров. На основе материалов, рассмотренных в разделе 1.3, предпочтение отдано Kubernetes.

Для удобства работы с Kubernetes был выбраны следующие вспомогательные компоненты:

— NFS в роли хранилища данных;

— Traefik в качестве реверс-прокси;

— GitLab Runner как инструмента CI/CD вместе с Helm;

— CertManager в роли средства атоматизации получения сертификатов.

Ниже приведен пример простого приложения в микросервисной архитектуре (рис. 2): пользователь взаимойдествует с фронтенд частью приложения, которая общается с бэкендом, а тот, в свою очередь, обменивается данными с базой данных.

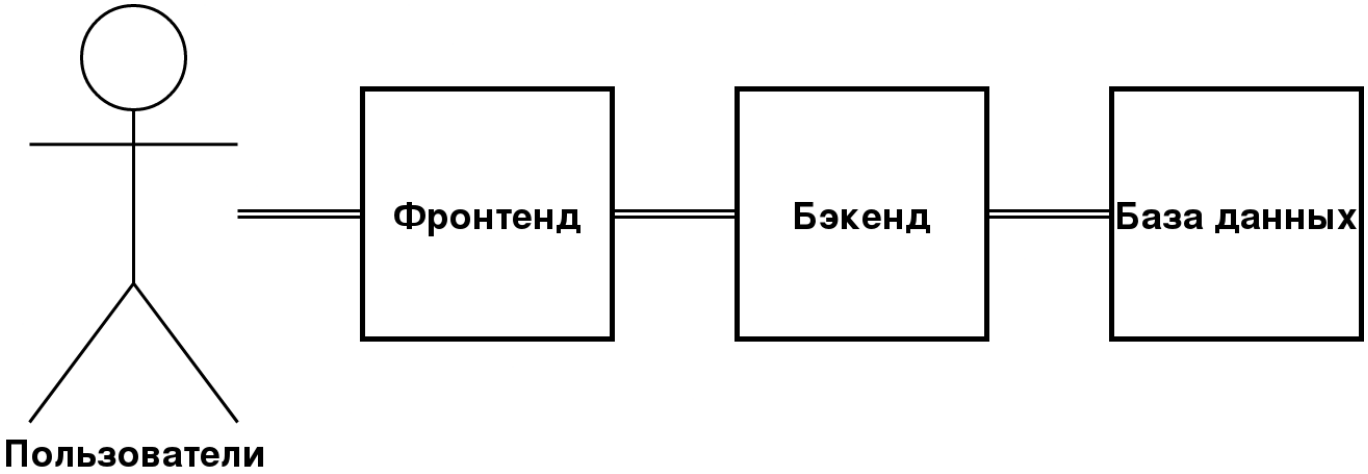

Рис. 2. Простое микросервисное приложение.

Далее изображена схема как трафик проходит от запроса пользователя до первого Pod, который будет обрабатывать трафик (рис. 3): в начале трафик попадает на Ingress Controller, который является реверс-прокси. Тот, в соответствии с правилами Ingress, перенаправляет их на Endpoints, которые созданы с помощью Service. А Endpoints являются списком IP адресов Pod, на которые нужно перенаправить трафик для его обработки.

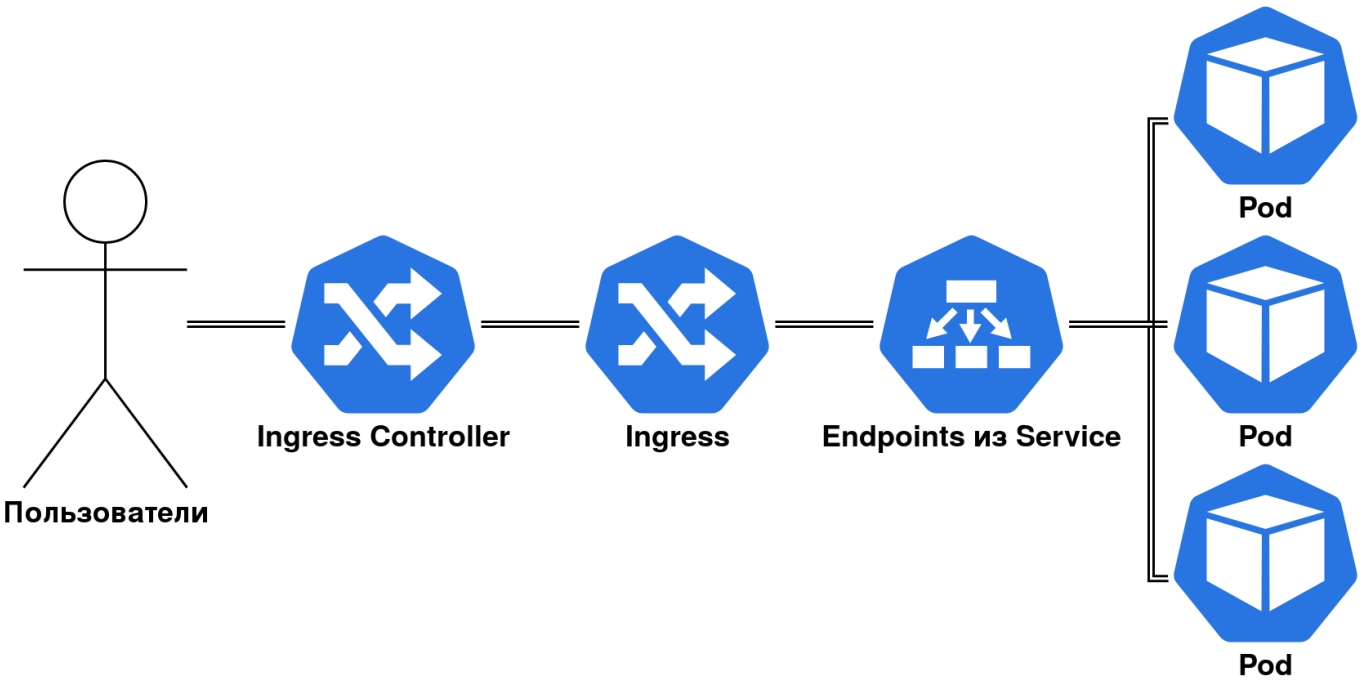

Рис. 3. Передача трафика от пользователя внутри Kubernetes.

Если совместить пример простого микросервисного приложения (рис. 2) и схему прохода трафика внутри Kubernetes, то получается более реалистичная иллюстрация происходящего (рис. 4):

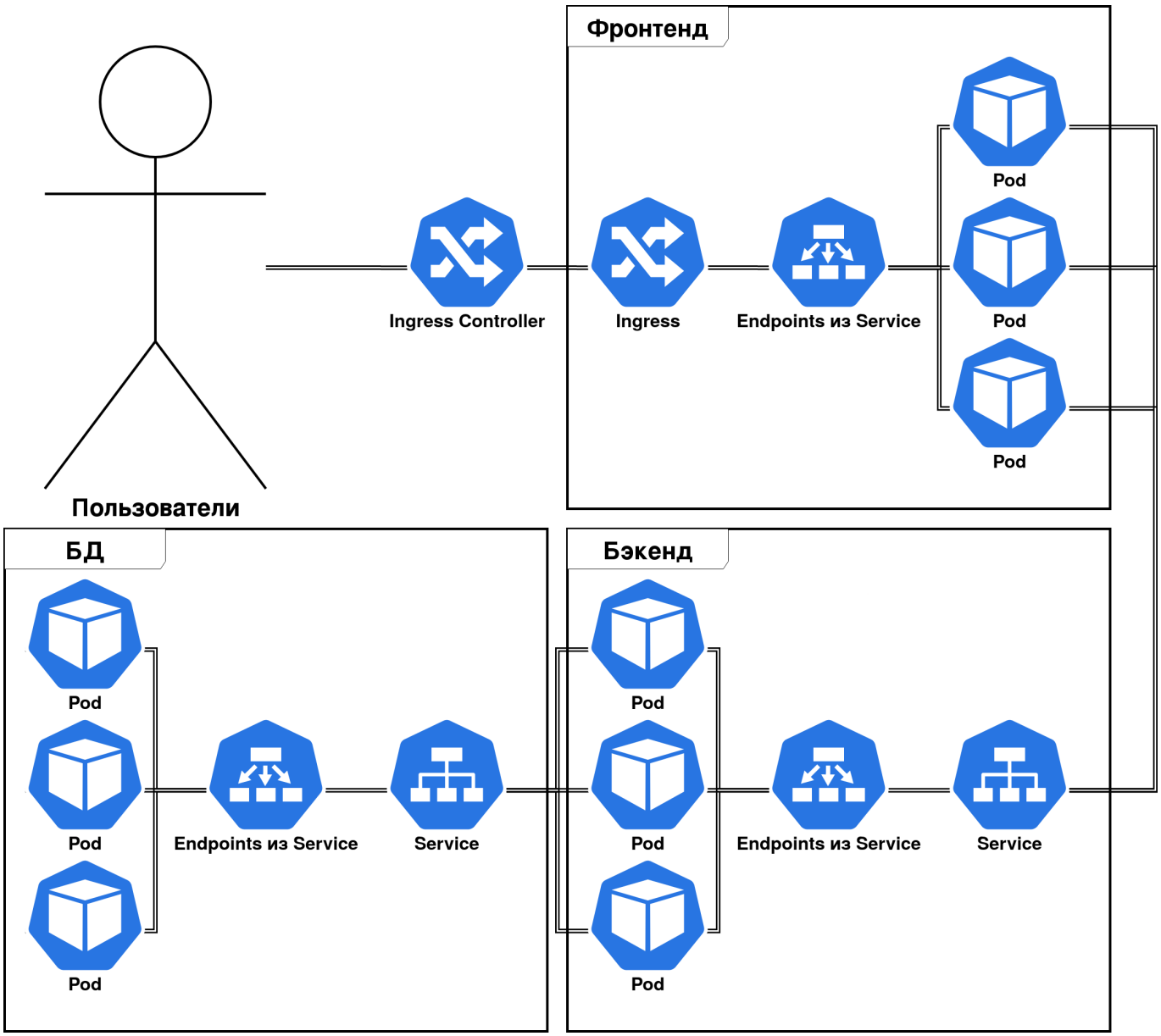

Рис. 4. Передача трафика от пользователя внутри Kubernetes для микросервисного приложения.

При первом приближении так выглядит взаимодействие разработчиков с новой архитектурой (рис. 5): разработчики взаимодействуют большую часть времени с системой контроля версий GitLab. Далее, в рамках CI/CD, собирается артефакт, который разворачивается в кластер Kubernetes с помощью Helm.

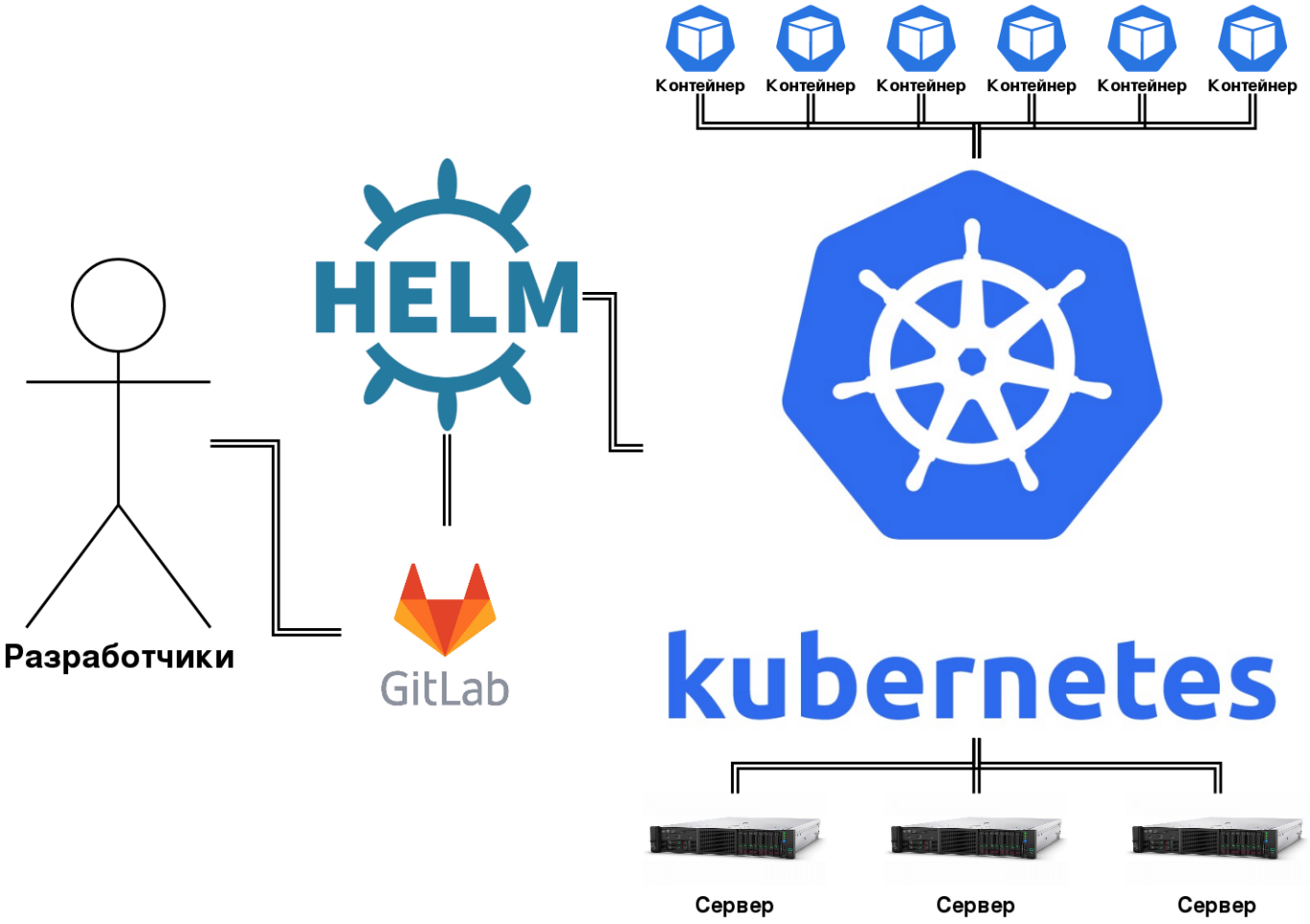

Рис. 5. Упрощенная схема взаимодействия разработчиков с архитектурой.

!!! Картинки с разными уровнями погружений и т.д. в новую архитектуру с пояснениями

2 Практическая часть работы

2.1 GitLab

Первым этапом было необходимо развренуть систему контроля версий. Ранее был выбран GitLab как комплексное решение данной задачи.

GitLab предоставляет на первый взгляд простое и удобное решение для собственного развертывания с помощью Docker **https://docs.gitlab.com/omnibus/docker/README.html**. Также, на официальном сайте есть примеры использования docker-compose **docker-compose** и docker swarm **docker swarm**. Недостатком этого решения является то, что несколько разных по своей сути микросервисов объединены в один контейнер Docker. Поэтому было решено разделить эти сущности на разные контейнеры. В итоге проделанной работы были разнесены, описаны как отдельные сущности и настроены:

— GitLab;

— Redis **Redis**;

— PostgreSQL **PostgreSQL**;

— Prometheus **Prometheus**;

— Grafana **Grafana**;

— Cadvisor **Cadvisor**;

— Node-exporter **Node-exporter**.

Детальнее можно ознакомится с конфигурацией GitLab и вспомогательных инстурментов в ПРИЛОЖЕНИИ ТАКОМ-ТО.

2.2 Kubernetes

Далее необходимо было развернуть кластер для контейнеров. Основным инструментом, решающим эту задачу был выбран Kubernetes. Официальным инструментом для выполнения этой задачи является

kubeadm, но он имеет ряд недостатков. Среди них стоит отметить сложность настройки нестандартных конфигураций, не говоря уже о сложности обновления кластера или подключения дополнительных нод. Поэтому был выбран один из самых популярных инструментов – kubespray. Этот инструмент, по своей сути – YAML-манифесты, с помощью которого описаны playbook для Ansible – программное решение для удаленного управления конфигурациями [25]. В свою очередь, tasks и roles в playbook используют kubeadm, но управлять им через данные абстракции в разы проще и удобнее. Также это позволяет иметь готовый артефакт, чтобы повторно развернуть кластер в другом месте или обновить его, внеся небольшие изменения.

Далее будут рассмотрены переменные, определенные в файле для развернутого кластера, отличающиеся от заданных по умолчанию:

а) Файл `inventory.ini`:

Данный файл кардинально отличается от оригинала, так как тот является лишь примером. В нем указаны в общей группе все инстансы, которые будут использоваться как ноды кластера. Среди них master и worker ноды, за названием каждой ноды должен быть указан ее IP адрес для подключения. Также master ноды должны быть перечислены в группе kube-master. Если etcd – высоконадёжное распределённое хранилище параметров конфигурации, задаваемых в форме ключ/значение [26], будет располагаться на master нодах, то их надо указать в группе etcd, иначе необходимо указать отдельные инстансы в общей группе под хранилище манифестов, и, соотвественно, указать их в группе хранилища. Worker ноды указываются в группе kube-node. Детальнее можно ознакомиться в ПРИЛОЖЕНИИ ТАКОМ-ТО.

б) Файл `group_vars/k8s-cluster/addons.yml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `dashboard_enabled: true`

Это позволяет получить базовый dashboard с состоянием кластера до момента поднятия полноценного мониторинга.

— `helm_enabled: true`

Эта опция устанавливает Helm – менеджер пакетов для Kubernetes, который позволяет разработчикам и операторам быстрее собирать, настраивать и развертывать приложения и сервисы в кластерах Kubernetes [27], вместе с кластером Kubernetes.

— `metrics_server_enabled: true`

Включает отдачу метрик.

— `ingress_nginx_enabled: false`

Это значение по умолчанию, но на первых этапах его стоит поменять на true, чтобы иметь в базовой установке кластера Ingress Controller на базе nginx.

— `ingress_nginx_host_network: true`

Данный параметр не будет задействован без предыдущего, но его стоит выставить в значение true, чтобы при включении базового контроллера он был работоспособен за счет обработки трафика с хостовой ноды.

— `ingress_nginx_namespace: «ingress-nginx»`

Здесь указано пространство в кластере, в котором будет находится контроллер. Этот параметр стоит раскомментировать, чтобы четко задать пространство, не предоставляя возможности разместить контроллер в пространстве по умолчанию или же служебном пространстве.

— `ingress_nginx_insecure_port: 80`

Порт, на который должны приходить HTTP-данные.

— `ingress_nginx_secure_port: 443`

Порт, на который должны приходить HTTPS-данные.

в) Файл `group_vars/k8s-cluster/k8-cluster.yml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `kube_version: v1.19.6`

На момент развертывания кластера последняя из стабильных версий kubelet.

— `kube_service_addresses: 10.10.0.0/16`

Более широкая по диапазону адресов сеть, нежели предложенная по умолчанию. Также необходимо отслеживать, чтобы виртуальные сети кластера не пересекались с сетями на хостовых нодах.

— `kube_pods_subnet: 10.15.0.0/16`

Аналогично предыдущему параметру, указана более широкая сеть.

— `cluster_name: cluster.itsoft`

Задание имени кластера, что также влияет на внутренние DNS-имена.

— `podsecuritypolicy_enabled: true`

Включение PodSecurityPolicy в кластере, что позволит настроить ограничения работы контейнеров и других сущностей для большей безопасности.

— `kubeconfig_localhost: true`

Установка служебной утилиты на машину, с которой производится развертывание кластера.

— `kubectl_localhost: true`

Установка служебной утилиты для управления кластером на машину, с которой производится развертывание кластера.

г) Файл `group_vars/k8s-cluster/k8-net-calico.yml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `calico_ip_auto_method: “interface=ens3”`

Опция, позволяющая сетевому плагину для кластера автоматически определить, какую сеть использовать для работы между нодами, на основе указанного сетевого интерфейса.

д) Файл `group_vars/etcd.yml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `etcd_memory_limit: 0`

Такое задание значения параметра снимает ограничения по памяти на хранилище манифестов etcd, что критично для его работы.

е) Файл `group_vars/all/docker.yml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `docker_storage_options: -s overlay2`

Так как на хостах используется свежая версия ядра ОС, то желательно использование более современного драйвера хранения данных для Docker.

После адаптации конфигурационных файлов под задачи и доступные вычислительные ресурсы, необходимо запустить playbook `cluster.yml`. При дальнейших обновлениях кластера необходимо использовать файл `upgrade-cluster.yml`.

2.3 NFS

Следующим этапом стало подключение системы хранения данных для контейнеров. Для этого была создана NFS – сетевая файловая система. Она подключена с помощью проекта NFS Subdir External Provisioner для кластера Kubernetes. Проект также пришлось адаптировать под созданные хранилища отдельно на SSD и HDD:

а) Файл `deploy/helm/values.yaml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

На месте примера с одной подключаемой NFS была переписана структура YAML-манифеста, которая теперь включает подразделы ssd и hdd, в которых уже, в свою очередь, указывается IP адрес сервера, путь до директории на сервере и опции монтирования.

б) Файл `deploy/helm/templates/deployment.yaml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

Из-за подключения двух NFS у Storage Provisioner для Kubernetes необходимо было создать, соотвественно, два контейнера: один для хранилища на SSD, второй для HDD. В связи с этим были продублирован раздел с контейнером, изменены их имена и метки, подправлены обращения к переменным в `deploy/helm/values.yaml`, и добавлены необходимые volumes для монтирования.

в) Файл `deploy/helm/templates/storageclass.yaml` (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

Также, как и в `deploy/helm/templates/deployment.yaml`, были скорректированы обращения к переменным в `deploy/helm/values.yaml`, продублирован StorageClass, поправлены имена во избежание коллизий.

После выполнения подготовки и настройки конфигурации под имеющиеся NFS, необходимые сущности Kubernetes разворачиваются с помощью Helm по инструкции проекта. Также, после развертывания проекта, необходимо написать YAML-манифесты для создания набора тестовых экземпляров сущностей Kubernetes, чтобы проверить работу хранилища: PersistentVolumeClaim, обращающегося к необходимому StorageClass, несколько Pod с PersistentVolume, который обращается к указанному ранее PersistentVolumeClaim, один из которых записывает в него данные, а другой позволяет их прочесть (ПРИЛОЖЕНИЕ ТАКОЕ-ТО).

2.4 Traefik

Далее необходимо было настроить Ingress Controller для промышленного использования, то есть обработки сетевого трафика. В

качестве этого инструмента был выбран Traefik. Он, в сравнении с предлагаемым по умолчанию nginx, является намного более современным, лишенным некоторых ощутимых недостатков, которые присущи nginx. Его настройка также заслуживает внимания:

Файл values.yaml (ПРИЛОЖЕНИЕ ТАКОЕ-ТО):

— `ingressClass[enabled]: true`

Этот параметр означает, что будет создана сущность IngressClass в кластере Kubernetes для того, чтобы он корректно увидел новый Ingress Controller.

— `ingressClass[isDefaultClass]: true`

Задание Traefik как Ingress Controller по умолчанию.

— `pilot[enabled]: true`

Эта опция включает возможность управления и мониторинга активности Traefik посредством веб-интерфейса. За ней следует еще одна, содержащая токен.

— `volumes`

Этот раздел параметров указывает сущности Kubernetes, которые необходимо подключить к Pod Traefik. На данном этапе это используется, чтобы подключить wildcard сертификаты для сайтов.

— `ports[web][hostPort]: 80`

Это значение позволяет включить проброс сетевого трафика с 80-го порта инстанса, на котором будет находиться Pod Traefik на заданный по умолчанию порт самого Pod Traefik.

— `ports[web][redirectTo]: websecure`

Эта опция включает перенаправления трафика с HTTP на HTTPS.

— `ports[websecure][hostPort]: 443`

Аналогично подобной опции в разделе `web`, позволяет проброс трафика с хостовой машины внутрь Pod Traefik.

— `ports[websecure][tls][enabled]: true`

Эту опцию необходимо включать для того, чтобы работали сертификаты для HTTPS. Далее, в разделе tls, указываются пути до сертификатов и обслуживаемые доменные имена.

— `service[enabled]: false`

Так как у нас Traefik является входным узлом, то для него не требуется сущность Service Kubernetes.

— `persistence[enabled]: true`

Использование PerstistentVolumeClaim Kubernetes для хранения TLS сертификатов. Далее также необходимо указать корректный StorageClass, который был создан ранее.

— `podSecurityPolicy[enabled]: true`

Включение аспектов безопасности Pod Traefik.

— `resources`

Этот раздел аргументов позволяет задать минимальные требования по ресурсам для запуска Traefik, а также его лимиты, за которые он не сможет выйти по потреблению.

— `nodeSelector`

С помощью сущности NodeSelector можно указать, на каком хосте или группе хостов должен оказаться конкретный Pod. В данном случае, это позволяет расположить Pod Traefik именно там, куда проброшен трафик из Интернета.

— `tolerations`

С помощью данной сущности Kubernetes можно обойти какие-либо ограничения. В текущем случае, это ограничение на планирование запуска Pod на master ноде.

2.5 CI/CD

После этого была начата разработка тестового проекта для развертывания в кластере с помощью CI/CD. За основу был взят минимальный веб-сервер на языке программирования Go – язык программирования, начало которого было положено в 2007 году сотрудниками компании Google [28], который отдавал статичную страницу. Далее был написан Dockerfile – описание правил по сборке образа, в котором первая строка указывает на базовый образ [4], с командами для сборки и запуска веб-сервера. Первая его часть основан на Docker Image `golang:1.8-alpine`. Далее код копируется внутрь Docker Image посредством директивы texttt`ADD` и компилируется с помощью директивы `RUN`. После этого берется второй Docker Image `alpine:latest`, в который переносится скомпилированный веб-сервер директивой `COPY`. Также объявляется порт для прослушивания трафика директивой `ENV`. И последней директивой `CMD` запускается веб-сервер.

Далее было необходимо написать Helm-chart для развертывания контейнера Docker и всех необходимых для его работы сущностей Kubernetes в кластере. В качестве ключевых файлов выступают `.helm/values.yaml` и `.helm/Chart.yaml`, в которых содержатся:

— Название проекта.

— Описание проекта.

— Версия проекта.

— Версия используемого API [29].

— Название Docker Image.

— Тег Docker Image.

— Политика скачивания образа.

— Количество копий образа, которое должно быть запущено.

— Порт, на котором веб-сервер образа слушает трафик.

На основе этих переменных все остальные файлы в директории `.helm/templates` будут описывать сущности кластера Kubernetes:

— `helm/templates/deployment-app.yaml`

— `helm/templates/service-app.yaml`

— `helm/templates/ingress.yaml`

Далее был написан файл `.gitlab-ci.yml`, который является YAML-манифестом для описания этапов CI/CD для развертывание проекта в кластере. Он состоит из трех этапов:

— `build`.

— `push`.

— `deploy`.

Этап `build` выполняется в одну команду `docker build` с указанием тега самим GitLab Runner.

Этап `push` требует предварительной аутентификации в Docker Registry – зарезервированный сервер, используемый для хранения docker-образов [4], для этого был взят пример из официальной документации по GitLab. После этого выполняется `docker push`.

Этап `deploy`, относящийся к CD уже намного сложнее. Для этого требуется задать в GitLab API URL кластера Kubernetes, а также его токен. После чего в начале выполнения этапа мы задаем эти переменные в окружение с помощью `kubectl`, после чего происходит проверка на то, существует ли уже данный проект в кластере. Если проект существет, то будет выполнена команда `helm upgrade`, в противном случае будет

выполнена команда `helm install`. Детальнее эти вещи можно увидеть в ПРИЛОЖЕНИЕ ТАКОМ-ТО.

2.6 Certs

Последним этапом развертывания кластера стало автоматическое получение сертификатов для проекта. Для этого был установлен проект CertManager в кластер Kubernetes. После чего был доработан Helm-chart, в котором появились сущности Certificate и Issuer, которые взаимодействуют с центром сертификации Let's Encrypt через CertManager. Их содержимое отображено в ПРИЛОЖЕНИИ ТАКОМ-ТО.

2.7 Портал для автоматизации развертывания проекта

После развертывания кластера необходимо было автоматизировать процесс выдачи доступа к кластеру инструментам CI/CD уникально для каждого проекта. Для этого был разработан веб-портал. Аутентификация на нем происходит посредством единого аккаунта НИУ ВШЭ. Авторизация работает на основе данных из личного кабинета проектной деятельности МИЭМ НИУ ВШЭ. В итоге получается схема, изображенная ниже (рис. 6):

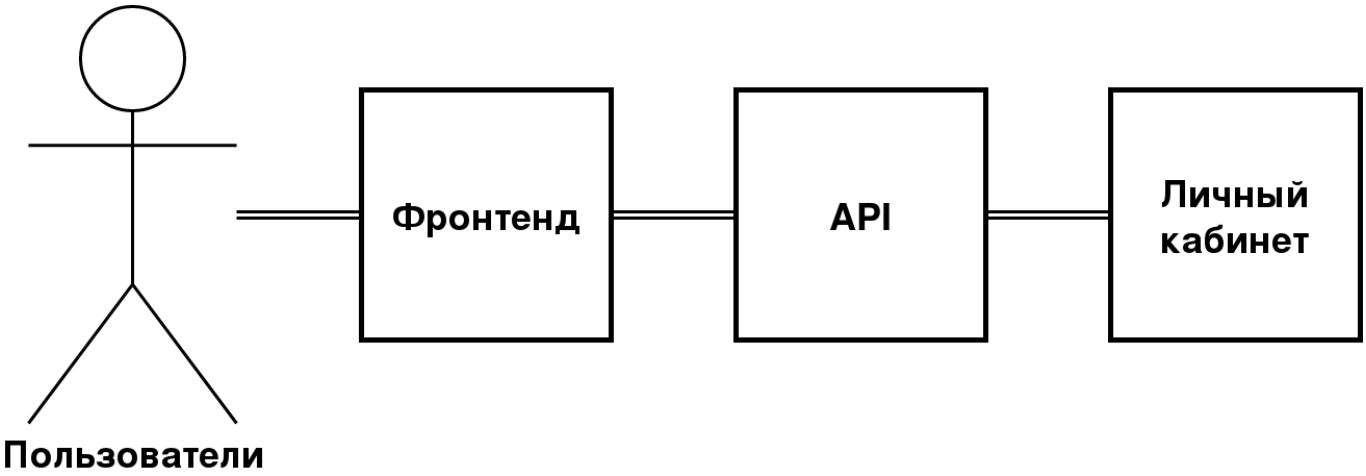

Рис. 6. Упрощенная схема веб-портала

Детальнее изучить работу веб-портала можно в ПРИЛОЖЕНИИ ТАКОМ-ТО.

ЗАКЛЮЧЕНИЕ

ОПРЕДЕЛЕНИЯ

Ansible — программное решение для удаленного управления конфигурациями [25].

Application Programming Interface — описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой [29].

Block Input/Output — обращение на считывание и запись с хранителей данных.

Central Processing Unit — процессор.

Certificate — blablabla.

CertManager — blablabla.

Continuous Integration и Continuous Delivery или Deployment — этапы непрерывной интеграции разработок и непрерывной доставки кода вплоть до промышленного использования [9].

Control group — управление (контроль) групп.

Dashboard — blablabla.

DLL Hell — сбой, возникающий, когда одна часть программного обеспечения ведет себя не так, как ожидалось второй частью программного обеспечения, что в некотором роде «зависит» от действия первого [3].

Docker — программное обеспечение для автоматизации развертывания и управления приложениями в среде виртуализации на уровне операционной системы [5].

Docker Registry — зарезервированный сервер, используемый для хранения docker-образов [4].

Dockerfile — описание правил по сборке образа, в котором первая строка указывает на базовый образ [4].

Etcd — высоконадёжное распределённое хранилище параметров конфигурации, задаваемых в форме ключ/значение [26].

Git — одна из систем контроля версий [18].

GitLab Runner — blablabla.

Go — язык программирования, начало которого было положено в 2007 году сотрудниками компании Google [28].

Helm — менеджер пакетов для Kubernetes, который позволяет разработчикам и операторам быстрее собирать, настраивать и развертывать приложения и сервисы в кластерах Kubernetes [27].

Image или образ контейнера — файл, включающий зависимости, сведения, конфигурацию для дальнейшего развертывания и инициализации контейнера [4].

Ingress Controller — blablabla.

Issuer — blablabla.

Kafka — blablabla123.

kubeadm — blablabla.

kubectl — blablabla.

kubelet — blablabla.

Kubernetes — blablabla.

kubespray — blablabla.

Let's Encrypt — blablabla.

master — blablabla.

Memory — память.

Mount — монтирование, например дисков как директорий. Способ подключения различных хранителей данных в OS GNU/Linux.

Multi stage approval — blablabla.

Namespaces — пространства имен.

Networking или network — сеть.

nginx — blablabla.

NodeSelector — blablabla.

PersistentVolume — blablabla.

PersistentVolumeClaim — blablabla.

playbook — blablabla.

Pod — blablabla.

PodSecurityPolicy — blablabla.

Process ID — идентификатор процесса.

Quorum approval — blablabla.

Resource management — управление и ограничение ресурсов машины.

roles — blablabla.

Service — blablabla.

Shell — средство для запуска других программ в ОС GNU/Linux.

Statefull application — blablabla.

Stateless — blablabla.

Storage Provisioner — blablabla.

StorageClass — blablabla.

tasks — blablabla.

TLS — blablabla.

Traefik — blablabla.

Uniform Resource Locator — blablabla.

User — пользователь.

volumes — blablabla.

Wildcard сертификат — blablabla.

worker — blablabla.

YAML — blablabla.

Виртуальная машина — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (target — целевая, или гостевая платформа) и исполняющая программы для target-платформы на host-платформе (host — хост-платформа, платформа-хозяин) или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы [1].

Гипервизор — программа или аппаратная схема, обеспечивающая или позволяющая одновременное, параллельное выполнение нескольких операционных систем на одном и том же хост-компьютере [2].

Инстанс — экземлпяр объекта.

Микросервисная архитектура — blablabla.

Реверс-прокси — blablabla.

Система контроля версий — blablabla.

Средства непрерывной интеграции и доставки артефактов — blablabla.

Токен — blablabla.

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

VM — virtual machine (виртуальная машина).

OS или ОС — операционная система.

ПО — программное обеспечение.

PID — Process ID.

Cgroup — control group.

CPU — Central Processing Unit.

I/O — input and output.

CI/CD — Continuous Integration и Continuous Delivery или Deployment.

K8s — Kubernetes.

БД — база данных.

СУБД — система управления базами данных.

IP — blablabla.

HTTP — blablabla.

HTTPS — blablabla.

DNS — blablabla.

NFS — blablabla.

SSD — blablabla.

HDD — blablabla.

API — Application Programming Interface.

URL — blablabla.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Wikipedia. Виртуальная машина — Википедия. — 2021. — Режим доступа: https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D0%B0&stable=1 (дата обращения: 07.05.2021).

2. Wikipedia. Гипервизор — Википедия. — 2021. — Режим доступа: https://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D0%B2%D0%B8%D0%B7%D0%BE%D1%80 (дата обращения: 07.05.2021).

3. Dick Stephanie, Volmar Daniel. DLL hell: software dependencies, failure, and the maintenance of microsoft windows // IEEE Annals of the History of Computing. — 2018. — Vol. 40, no. 4. — P. 28–51.

4. eternalhost. Что такое Docker и зачем он нужен – примеры использования. — 2020. — Режим доступа: https://eternalhost.net/blog/razrabotka/chto-takoe-docker (дата обращения: 10.05.2021).

5. Демидова ТС, Соболев АА. Использование Docker для развёртывания вычислительного программного комплекса // Информационно-телекоммуникационные технологии и математическое моделирование высокотехнологичных систем. — 2019. — P. 432–435.

6. Kim Clark Callum Jackson. What are the benefits of containers? Part 1 – IBM Developer. — 2020. — Access mode: https://developer.ibm.com/components/kubernetes/articles/true-benefits-of-moving-to-containers-1/ (online; accessed: 10.05.2021).

7. Kim Clark Callum Jackson. What are the benefits of containers? Part 2 – IBM Developer. — 2020. — Access mode: https://developer.ibm.com/components/kubernetes/articles/true-benefits-of-moving-to-containers-2/ (online; accessed: 10.05.2021).

8. Jain Rahul. Битва Jenkins и GitLab CI/CD / Блог компании RUVDS.com / Хабр. — 2020. — Режим доступа: https://habr.com/ru/company/ruvds/blog/522334/ (дата обращения: 10.05.2021).

9. Stolyarov Dmitry. Лучшие практики CI/CD с Kubernetes и GitLab (обзор и видео доклада) / Блог компании Флант / Хабр. — 2017. — Режим доступа: https://habr.com/ru/company/flant/blog/345116/ (дата обращения: 10.05.2021).

10. Container orchestration engines: a thorough functional and performance comparison / Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi et al. // ICC 2019-2019 IEEE International Conference on Communications (ICC) / IEEE. — 2019. — P. 1–6.

11. Nüst Daniel, Sochat Vanessa, Marwick Ben et al. Ten simple rules for writing Dockerfiles for reproducible data science. — 2020.

12. Recommendations for the packaging and containerizing of bioinformatics software / Bjorn Gruening, Olivier Sallou, Pablo Moreno et al. // F1000Research. — 2018. — Vol. 7.

13. VASS TIBOR. Intro Guide to Dockerfile Best Practices - Docker Blog. — 2020. — Режим доступа: https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/ (дата обращения: 10.05.2021).

14. Wang Yujing, Ma Darrel. Developing a process in architecting microservice infrastructure with Docker, Kubernetes, and Istio // arXiv preprint arXiv:1911.02275. — 2019.

15. Indrasiri Kasun, Siriwardena Prabath. Microservices for the enterprise // Apress, Berkeley. — 2018.

16. Naik Nitin. Building a virtual system of systems using docker swarm in multiple clouds // 2016 IEEE International Symposium on Systems Engineering (ISSE) / IEEE. — 2016. — P. 1–3.

17. Игрычев Алексей. Пакетный менеджер для Kubernetes — Helm: прошлое, настоящее, будущее / Блог компании Флант / Хабр. — 2018. — Режим доступа: https://habr.com/ru/company/flant/blog/417079/ (дата обращения: 10.05.2021).

18. Имя Отображаемое. Работа с распределенной системой контроля версий Git на примере GitHub / Хабр. — 2019. — Режим доступа: https://habr.com/ru/post/451662/ (дата обращения: 10.05.2021).

19. Stolyarov Dmitry. Наш опыт с Kubernetes в небольших проектах (обзор и видео доклада) / Блог компании Флант / Хабр. — 2017. — Режим доступа: https://habr.com/ru/company/flant/blog/331188/ (дата обращения: 10.05.2021).

20. Шурупов Дмитрий. Практики Continuous Delivery с Docker (обзор и видео) / Блог компании Флант / Хабр. — 2017. — Режим доступа: https://habr.com/ru/company/flant/blog/322686/ (дата обращения: 10.05.2021).

21. Stolyarov Dmitry. Базы данных и Kubernetes (обзор и видео доклада) / Блог компании Флант / Хабр. — 2018. — Режим доступа: https://habr.com/ru/company/flant/blog/431500/ (дата обращения: 10.05.2021).

22. Stolyarov Dmitry. Мониторинг и Kubernetes (обзор и видео доклада) / Блог компании Флант / Хабр. — 2018. — Режим доступа: https://habr.com/ru/company/flant/blog/412901/ (дата обращения: 10.05.2021).

23. Даниил. Kubernetes (k8s) + Helm + GitLab CI/CD. — 2018. — Режим доступа: https://habr.com/ru/post/422493/ (дата обращения: 10.05.2021).

24. Игрычев Алексей. Проблема «умной» очистки образов контейнеров и её решение в werf / Блог компании Флант / Хабр. — 2020. — Режим доступа: https://habr.com/ru/company/flant/blog/522024/ (дата обращения: 10.05.2021).

25. Давлеткалиев Рахим. Пособие по Ansible / Хабр. — 2016. — Режим доступа: https://habr.com/ru/post/305400/ (дата обращения: 10.05.2021).

26. GitHub. GitHub - etcd-io/etcd: Distributed reliable key-value store for the most critical data of a distributed system. — 2021. — Access mode: https://github.com/etcd-io/etcd (online; accessed: 10.05.2021).

27. Amber. Основы работы с Helm, пакетным менеджером Kubernetes | 8HOST.COM. — 2018. — Режим доступа: https://www.8host.com/blog/osnovy-raboty-s-helm-paketnym-menedzherom-kubernetes/ (дата обращения: 10.05.2021).

28. Наливайко Александр. Golang: основы для начинающих. — 2017. — Режим доступа: https://tproger.ru/translations/golang-basics/ (дата обращения: 10.05.2021).

29. Wikipedia. API — Википедия. — 2021. — Режим доступа: https://ru.wikipedia.org/wiki/API (дата обращения: 10.05.2021).

ПРИЛОЖЕНИЕ А

ИНСТРУКЦИЯ ДЛЯ ПОЛЬЗОВАТЕЛЯ

Проект должен являться микросервисным приложением, желательно разделенным на несколько Docker образов.

Для проверки работоспособности на собственных вычислительных мощностях с помощью docker-compose, необходимо:

— Установить систему контейнеризации Docker и менеджер контейнеров Docker-compose;

— Подготовить директорию с исходным кодом проекта;

— Написать docker-compose манифест (пример – **https://docs.docker.com/compose/gettingstarted/**);

— Запустить проект с помощью `docker-compose up`;

— Проверить работоспособность каждого модуля проекта.

Далее необходимо подготовить проект для развертывания в кластере. Для этого необходимо:

— Получить данные для авторизации в кластере проекта в GitLab;

— Подготовить манифесты для развертывания проекта в кластере;

— Настроить репозиторий с помощью полученных данных для авторизации в кластере;

— Запустить процесс CI/CD посредством обновления данных в репозитории.

Получить данные для авторизации в кластере проекта в GitLab можно посредством использования веб-портала АДРЕС???. На нем необходимо пройти авторизацию через единую систему авторизации НИУ ВШЭ, после чего выбрать необходимый проект и последовательно следовать инструкции на портале.

Для упрощения подготовки манифестов для развертывание проекта создан шаблон, в котором находится простое веб-приложение на языке программирования Go и манифесты для сборки и развертывания проекта. Шаблон доступен по адресу **https://git.miem.hse.ru/423/demo-projects**. В нем содержится:

— Исходный код проекта – `main.go`;

— Dockerfile для упаковки приложения в образ – `Dockerfile`

— Манифест для сборки и запуска процесса развертывания – `.gitlab-ci.yml`

— Директория с манифестами для развертывания проекта в кластер – `.helm`

Для оптимизации написания собственного Dockerfile необходимо изучить материал с примерами и лучшими практиками **https://docs.docker.com/develop/develop-images/dockerfile_best-practices/**

Манифест для сборки и запуска процесса развертывания должен состоять из трех этапов:

— `build`

— `push`

— `deploy`

На этапе `build` выполняется сборка образов Docker по инструкциям из Dockerfile. В случае, если необходимо собрать более одного образа Dockerfile, стоит расположить исходный код и Dockerfile к нему в отдельной поддиректории. И для каждого Dockerfile скорректировать инструкцию данного этапа, начинающуюся с `docker build`.

На этапе `push` выполняется отправка образов в Docker Registry. В случае, если на предыдущем этапе было собрано несколько образов, их все

необходимо отправить, склонировав и скорректировав инструкцию, начинающуюся с `docker push`, несколько раз.

На этапе `deploy` выполняется развертывание образа в кластер. Он основан на содержимом директории `.helm`. В случае, когда было собрано более одного образа, необходимо склонировать и скорректировать перегрузку переменных для образов согласно содержимому файла `.helm/values.yaml`. Эти команды начинаются со слов `–set`

Далее необходимо взять за основу директорию `.helm` из шаблона и скорректировать ее содержимое под реальный проект. В файле `.helm/Chart.yaml` нужно корректно задать следующие значения:

— `description`

— `name`

— `version`

После этого требует корректировки файл `.helm/values.yaml`. Его стилистка предполагает, что на каждый микросервис необходимо выделить свою подгруппу значений, как это сделано с `app`. Обязательными значениями являются:

— `repository` – можно поставить пустое значение в кавычках, так как это значение должно быть перегружено в GitLab CI/CD;

— `tag` – можно поставить пустое значение в кавычках, так как это значение должно быть перегружено в GitLab CI/CD;

— `pullPolicy` – предпочтительно значение `Always`, чтобы образ точно скачался при новом развертывании;

— `replicaCount` – количество экземлпяров образа для запуска. Стандартное значение – `1`;

— `port` – сетевой порт, на котором в образе поднят сервер.

Далее, под каждый группу значений нужно создать или скорректировать манифесты сущностей Kubernetes. Для развертывания образа необходима сущность Deployment, шаблон которой описан в `.helm/templates/deployment-app.yaml`. Для возможности взаимодействия с образом Docker по внутренней сети, необходима сущность Service – `.helm/templates/service-app.yaml`. Для обработки трафика из внешней сети нужна сущность Ingress – `.helm/templates/ingress.yaml`.

Последний этап подготовки проекта - настройка репозитория с помощью полученных данных для авторизации в кластере. Для этого необходимо, согласно инструкции с портала, на котором были получены данные, ввести их в настройках проекта в GitLab.

ПРИЛОЖЕНИЕ Б

РУКОВОДСТВО АДМИНИСТРАТОРА

Как обновить GitLab, K8s, мб еще что

ПРИЛОЖЕНИЕ В

ДОКУМЕНТАЦИЯ РАЗРАБОТЧИКА

Ознакомиться со всей вкр и другими доками