МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ
имени Гумарбека Даукеева

Кафедра «Телекоммуникационные сети и системы»
Специальность: 6М071900 «Радиотехника, электроника и телекоммуникации»

ДОПУЩЕН К ЗАЩИТЕ
Зав. кафедрой
PhD, доцент Темырканова Э.К.
(ученая степень, звание, ФИО)

_____
(подпись)
«_____»_____2020 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
пояснительная записка

на тему: «Система оркестрации контейнеров - Kubernetes»

Магистрант: Бекбосынов К.Д. _____ _____ группа МРЭТ 18-1
(Ф.И.О.) (подпись)
Руководитель: к.т.н., профессор _____ Байкенов А.С
(ученая степень, звание (подпись) (Ф.И.О.)

Рецензент_____ _____ _____
(ученая степень, звание) (подпись) (Ф.И.О.)

Консультант по ВТ к.т.н., профессор _____ Байкенов А.С.___
(ученая степень, звание) (подпись) (Ф.И.О.)

Нормоконтроль: _к.т.н., профессор _____ Байкенов А.С.
(ученая степень, звание) (подпись) (Ф.И.О.)

Алматы 2020

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН
Некоммерческое акционерное общество
АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ
имени Гумарбека Даукеева

Институт Космической Инженерии и Телекоммуникаций

Специальность: 6М071900 «Радиотехника, электроника и телекоммуникации»

Кафедра: «Телекоммуникационные сети и системы»

ЗАДАНИЕ
на выполнение магистерской диссертации

Магистранту Бекбосынову Кымбату Дуйсембековичу
(фамилия, имя, отчество)
Тема диссертации «Система оркестрации контейнеров - Kubernetes »

Утверждена Ученым советом университета №122 от «25» октябрь 2018
Срок сдачи законченной диссертации «25»_мая 2020г.
Цель работы состоит в разработке инфраструктурных решений на базе фреймворка Kubernetes и сопутствующих систем как альтернатива классической архитектуре в угоду автоматизации.

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации:
1. Основные направления развития и подходы применяемые для построения инфраструктуры
2. Базовые компоненты Kubernetes и сторонние решения интегрируемые с ним
3. Исследование аспекта сетевого взаимодействия и диагностики с помошью Istio
4 DevOps подходы в автоматизации инфраструктуры и его преимущества

Перечень графического материала (с точным указанием обязательных чертежей)
Рисунок 1.3 - Балансировка в Kubernetes на прокси
Рисунок 1.7 - Система мониторинга Prometheus
Рисунок 3.1- Deployment workflow
Рисунок 3.3- Выкат без простоя

Рекомендуемая основная литература

1. Marko Luksa, “Kubernetes in Action” – Manning, 2017 - 624c.
2. David Farley, Jez Humble, “Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation” – 2010 - 734c.

Г Р А Ф И К
подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
1. Информационный обзор согласно теме	05.10.2018	
2. Основные направления развития и подходы применяемые для построения инфраструктуры(теоритическа часть)	14.01.2019	
3. Исследование аспекта сетевого взаимодействия и диагностики с помошью Istio (исследовательская глава)	02.02.2019	
4. DevOps подходы в автоматизации инфраструктуры и его преимущества (исследовательская часть)	18.10.2020	
5. Анализ полученных экспериментальных данных	10.12.2020	

Дата выдачи задания_30 сентября 2018г._____

Заведующий кафедрой _____ (Темырканова Э.К.)
(подпись) (Ф.И.О.)

Научный
руководитель диссертации _____ (Байкенов А.С.)
(подпись) (Ф.И.О.)

Задание принял к исполнению
магистрант _____ (Бекбосынов К.Д.)
(подпись) (Ф.И.О.)

Аңдатпа

Бұл диссертациялық жұмыста Kubernetes-тің негізгі компоненттері қарастрылды. Kubernetes-пен үшінші тараптық Istio шешімімен интеграциялаудың және теңдестіретін функционалдылықты, қызметтерді және трафикті басқару алгоритмдерін біркелкі орналастыруды талдадық.

Инфрақұрылымды автоматтандыру, бақылау және тіркеу жұмыстарында бірқатар шешімдер сипатталған. Қазіргі заманғы өнеркәсіпте DevOps тәжірибесін қолданудағы бірқатар артықшылықтар мен кемшіліктерді сипаттады.

Аннотация

В этой диссертационной работе было рассмотрены основные компоненты Kubernetes. Исследовано стороннее решение Istio на интеграцию с Kubernetes, и анализ функционала балансировки, беспростойного развертывания сервисов и алгоритмы управления трафиком.

Описаны ряд решений в автоматизации инфраструктуры, мониторинга, журналирования. И описан ряд преемуществ и недостатков в применении практик DevOps в современной отрасли.

Annotation

This dissertation covered the core components of Kubernetes. We studied the third-party Istio solution for integration with Kubernetes, and the analysis of the balancing functionality, the uninterrupted deployment of services and traffic control algorithms.

A number of solutions are described in infrastructure automation, monitoring, and logging. And described a number of advantages and disadvantages in applying DevOps practices in the modern industry.

Содержание

Введение

Kubernetes – фреймворк для создания инфраструктуры на подобии PaaS, она оркестрирует docker контейнеры, распределяя их между узлов своего кластера и балансируя трафик на нужные endpoint-ы. Охватывая все большие датацентры распространения, Kubernetes развивается стремительно пополняя функционал и набор инструментов.

Google открыла проект Kubernetes в 2014 году. Kubernetes основывается на пятнадцатилетнем опыте, который Google имеет с масштабируемыми вычислениями, в сочетании с лучшими в своем классе идеями и практиками компании.

Kubernetes представляет собой черты нескольких платформ:

- Микросервисная платформа;
- Контейнерная платформа;
- Локальная облачная платформа.

Контейнер – главный элемент на которой строится система Kubernetes, исполняя главную роль в вычислительной необходимости. Сетевое взаимодействие производиться на выбор между несколькими вариантами: встроенных Kubernetes сетевых решений и внешних сетевых решений(в данной работе будет использована эта модель на базе Istio). Инфраструктуную эксплуатацию Kubernetes берет на себя что имеет сходство с PaaS

Под контейнерной платформой я имею ввиду docker контейнеры. Это базис на котором строиться сервисы. Суть контейнерной платформы в уменьшении потребляемых ресурсов не бизнес нагрузкой. Иногда можно добиться эффективности расхода ресурсов приближающейся к 99%. В большинстве случаев создается собственный образ на базе какого-то минимального образа и переделывается под свои нужды. Чтобы оставить только тот функционал, который необходим для выполнения задачи и более эффективного расходования ресурсов.

Kubernetes необходимо рассматривать как инструмент который строит среду для запуска сервисов, автоматизируя все внутренние процессы но оставляя интерфейс для взаимодействия с ним по API, конечно же разработан инструмент для работы с CLI для ручного взаимодействия и разного рода обертки как например helm шаблонизатор. То есть специалисту достаточно лишь однажды выбрать нужные параметры планируемого сервиса и строить кластер Kubernetes исходя из масштабов развития этого сервиса на ближайшие несколько лет

В Kubernetes ссылка на обекты реализовано с помощью меток. Проставляя метки сущьности обьединяються на логические группы что облегчает их поиск и обращение к ним. Аннотацией передаються дополнительные настройки на обьект в манифесте создавая гибкий уровень абстракции для взаимодействия.

API Kubernetes защищен шифрованием RSA, между мастер узлом и клиентом обращающимся к интерфейсу. Гибкость API позволяет

интегрировать или разрабатывать новый функционал для Kubernetes. Как пример, операторы, сильно расширяющие стандартный функционал базовых возможностей. Написанные на языке программирования они представляют отдельные приложения, некоторые из них весьма популяры в промышленной эксплуатации: Prometheus, postgresql, helm и т.д.

Резюме

В этой диссертационной работе описывается базовый принцип работы системы Kubernetes и подходы применяемые DevOps практиками. Исследовательская работа затрагивает проблемы релиза сервисов и ее варианты реализации при помощи системы Istio (Envoy proxy), а так же автоматизацию, мониторинг, выявление и устранение проблем, и гибкую настройку маршрутизации. Хоть эти инструменты и подходы появились относительно недавно, но уже большое количество организаций используют их в промышленной эксплуатации (Google, Facebook, YouTube, Amazon, Ebay и т.д.) благодаря большой гибкости и экономичности этой реализации. А рынок труда выдвигает специалистам все новые требования (в сторону увеличения) необходимые для работы в современных условиях. Учитывая, что каждые два-пять лет появляются новые решения улучшающие нашу работу, из которых половина бесследно забывается, не выдерживая конкуренции. Специалисты вынуждены рассматривать и развивать навыки в новых средствах как DevOps.

Проработав на позиции инженера автоматизации более 10 лет, я вижу что Kubernetes и контейнерная технология стала третьим крупным прорывом в индустрии оркестрации сервисов. Первая – появление виртуальных машин? которая позволила разделить ресурсы одного физического сервера на виртуальные сервера с меньшими ресурсами. Преимущество данного метода заключается в получении полностью независимой виртуальной машины, ограниченной только зависимостями ресурсов гипервизора. Второе – появление контейнеров docker, стало большим толчком к переходу к микросервисной архитектуре и горизонтально масштабируемым системам. Контейнеры лишены большинства дублирующих функций виртуальных машин, за счет чего их полезное потребление ресурсов приближается к 100%. Третье – Kubernetes стал следующим необходимым компонентом для оркестрации контейнеров, не ограничиваясь только сервисами, но и предлагая возможности диагностики и самоуправления. То есть уменьшен человеческий фактор в жизни сервиса, а сотрудник лишь взаимодействует с Kubernetes с помощью манифестов декларативным способом.

Пакетный менеджер helm добавляет новый уровень абстракции взаимодействия с Kubernetes вместо манифестов, и дает более обширный набор инструментов для управления релизами сервисов. А в комбинации с конвейерами, журналами сервисов и статусами мониторинга общая картина решения выглядит самодостаточной и завершенной.

Надо понимать, что Kubernetes отвечает только за запуск контейнеров с определенными параметрами. Но так же нужно и отслеживать сервисы в контейнерах, собирать метрики, настроить безопасность и т.д. То есть без дополнительных обвесов Kubernetes стал бы крайне трудным в использовании. Таких сервисов для кластера существует большое множество, в работе я упоминаю только самые популярные из них, а так же привожу

примеры лучших практик их использования. И все больше на предприятиях возникают вопросы про навыки soft skills, где важную роль выделяют умению продуктивной коммуникации, не ограничиваясь только техническими навыками. Это ключевой пункт в DevOps практиках. Без продуктивной коммуникации в такой сложной системе как Kubernetes практически не возможно будет поддерживать сервисы в рабочем состоянии.

1 Архитектура Kubernetes

Традиционно сервисы развертывали непосредственно на узлах из исходных кодов или установочных пакетов. С появлением систем класса CMT(Configuration Management Tools) такие как puppet, ansible, chef этот процес стал более автоматизирован, но все еще требовал участия специалиста в планировании инфраструктуры и написании плейбуков(скриптов прогона для выполнения рутиных задач). Главным недостатком такого подхода считается низкая производительность.

В случае с Kubernetes подход развертывания отличается от традиционного где контейнеры представлены как неизменяемый обьект по концепции – Immutable infrastructure. И они разбрасываються между узлами кластера исходя из политик распределения, по умолчанию раскидываються равномерно между рабочими узлам кластера. Изоляция между сервисами выполняеться на уровне контейнерной виртуализации и сетевых модулей с возможностью шифрованиея до конечного пункта назначения. На уровне кластера существует множество абстраций по изоляции обьектов.

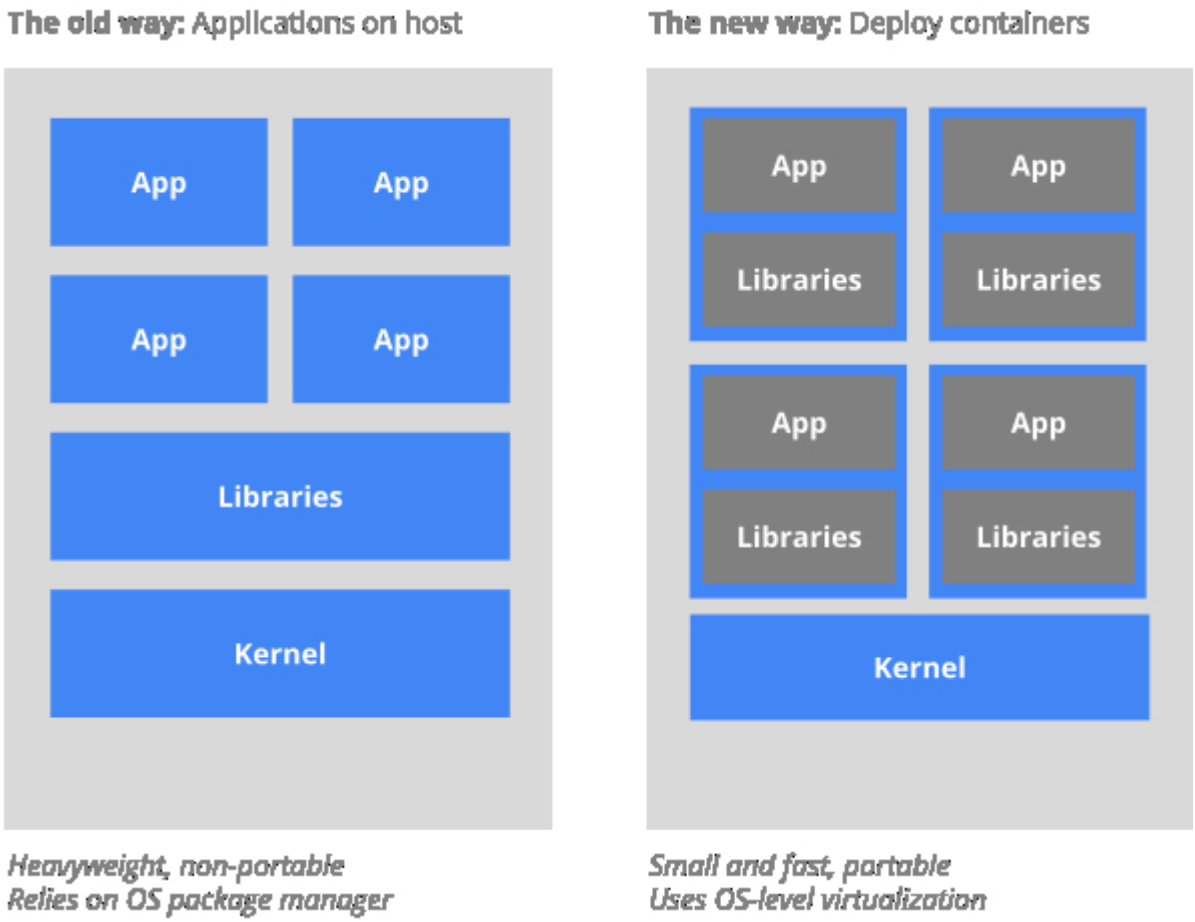

Рисунок 1.1 - Сравнение сервисов на контейнере и узле

Контейнеры представляют собой миниатюрную операционную систему с необходимыми зависимостями только для выполнения определенной задачи. По этой причине они занимают очень мало дискового пространства и их запуск длится небольшое время. По сути это классическая виртуальная машина но с маленьким размером, и выполняющий одну простую задачу. Такой подход называеться микросервисами и характеризуеться независимостью каждого сервиса от других.

Во время сборки образа контейнера процесс включает в себя скачивание операционной системы, установки необходимых програм и копировании файлов при необходимости. Собранный образ должен быть готовым для дальнейшего применения на различных средах вне зависимости от ситуации или давать возможность проброски недостающих компонентов через стандартный интерфейс взаимодействия как например переменные среды или монтируемый раздел в файловую систему узла где запушен данный контейнер. Таким образом контейнер становиться крайне гибким элементом.

Преимущества контейнера:

- Упрощение сервиса до отдельного элемента выделяя возможность гибкой настройки. Упрощение создание единицы сервиса и увеличение частоты выпуска версий.
- Непрерывная разработка, так называемая восьмерка DevOps где процес цикличной сборки и их выкатки происходит настолько часто что процес можно назвать непрерывным
- Обьединение процесса разработки сервиса и ее запуском в эксплатацию, с разделением зон ответственности при диагностике и поиске проблем.
- Мониторинг на уровне сервисов со сбором метрик и журналированием по отдельным сервисам контейнера.
- Воспроизводимость среды на любой платформе при условии наличия docker виртуализации.
- Переносимость и кросплатформенность как и в пункте выше, запуск возможен как на локальном ПК так и на облачном провайдере.
- Оркестрация, существует множество инструментов для управления контейнерами от базового функционала до польностью саморегулируемой инфраструктурой.
- Гибкость, позволяет разбить один большой проект на маленькие части, по принципу один минисервис это контейнер.
- Независимость каждого контейнера ограждает от пагубных действий соседнего контейнера.
- Эффективное потребление ресурсов гипервизора.

1.1 Обзор Pod

Pod является основным строительным блоком Kubernetes, самой маленькой и простой единицей в объектной модели Kubernetes, которую

создаете или развертываете. Pod представляет собой выполняемый процесс на кластере.

Pod инкапсулирует контейнер приложения (или, в некоторых случаях, несколько контейнеров), ресурсы хранения, уникальный сетевой IP и параметры, которые определяют, как должен работать контейнеры. Pod представляет собой единицу развертывания: один экземпляр приложения в Kubernetes, который может состоять либо из одного контейнера, либо из небольшого количества контейнеров, которые совместно используют ресурсы. Docker - это наиболее распространенное решение для контейнеризации, используемое в Kubernetes Pod, но Pod-ы поддерживают и другие контейнерные системы.

В кластере Kubernetes Pod использовуются двумя способами:

а) Pod, которые управляют одним контейнером. Модель «один контейнер-Pod» является наиболее распространенным случаем использования Kubernetes; в этом случае можно думать о Pod как об обертке вокруг одного контейнера, а Kubernetes управляет Pod, а не контейнерами напрямую;

б) Pod, которые запускают несколько контейнеров, которые должны работать вместе. Pod может инкапсулировать приложение, состоящее из нескольких контейнеров, расположенных совместно, которые тесно связаны друг с другом и должны совместно использовать ресурсы. Эти совместно расположенные контейнеры могут образовывать единую единицу обслуживания, один контейнер обслуживает файлы из общего тома для публики, в то время как отдельный контейнер «sidecar» обновляет эти файлы. Pod объединяет эти контейнеры и ресурсы хранения вместе как единый управляемый объект

Каждый Pod предназначен для запуска одного экземпляра данного приложения. Если необходимо масштабировать приложение по горизонтали (например, запускать несколько экземпляров), нужно использовать несколько Pod, по одному для каждого экземпляра. В Кубернете это обычно называют репликацией. Реплицированные Pod обычно создаются и управляются как группа абстракцией, называемой контроллером.

Pod в Kubernetes манифесте минимальный обьект для эксплуатации. При его создании определяеться образ и название, все остальные параметры настроек вторичны и кластер сам указывает как заполнить вторичные параметры как например узел где он будет запущен, ресурсы пробрасываемые в контейнер, количество контейнеров и т.д. Но так же есть системные параметры не доступные для изменения.

Обратите внимание, что группирование нескольких совместно расположенных и совместно управляемых контейнеров в одном Pod является относительно продвинутым вариантом использования. Этот шаблон следует использовать только в определенных случаях, когда контейнеры тесно связаны. Например, у может быть контейнер, который действует как веб-сервер для файлов в общем томе и отдельный контейнер «sidecar», который обновляет эти файлы из удаленного источника, как на следующей диаграмме:

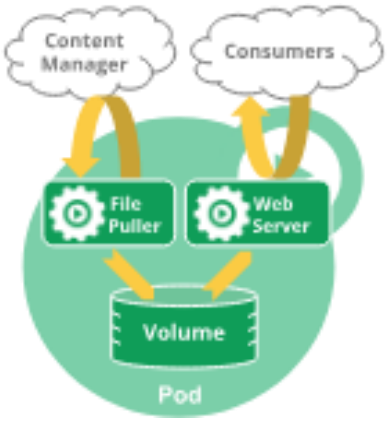

Рисунок 1.2 - Взаимодействие сервиса с ресурсом

Pods предоставляют два типа общих ресурсов для своих составных контейнеров: сеть и хранилище.

- сеть: каждому Pod присваивается уникальный IP-адрес. Каждый контейнер в Pod разделяет пространство имен в сети, включая IP-адрес и сетевые порты. Контейнеры внутри Pod могут связываться друг с другом с помощью localhost. Когда контейнеры в Pod взаимодействуют с объектами вне Pod, они должны координировать использование общих сетевых ресурсов (таких как порты);

- хранилище: Pod может указать набор разделяемых томов хранилища. Все контейнеры в Pod могут обращаться к общим томам, позволяя этим контейнерам обмениваться данными. Тома также позволяют сохранить постоянные данные в Pod, если один из контейнеров внутри должен быть перезапущен. Для персистентного хранилищя популярно использовать распределенную систему хранения Ceph с 3 узлами и 3 репликами данных.

Редко когда создаеться отдельные Pod непосредственно в Kubernetes, даже singleton Pods. Это связано с тем, что Pods разработаны как относительно эфемерные, одноразовые объекты. Когда Pod создается (напрямую или косвенно с помощью контроллера), он планируется запустить на узле в кластере. Pod остается в этом узле до тех пор, пока процесс не будет завершен, объект pod будет удален, pod высвобождает занимаемые ресурсы или это приведет к сбою узла.

Pod-ы сами по себе не самоизлечиваются. Если Pod назначается на аварийный узел, или если сама операция планирования завершается неудачно, Pod удаляется; Аналогично, Pod не запуститься из-за нехватки ресурсов или аварийности узла. Kubernetes использует абстракцию более высокого уровня, называемую контроллером, которая обрабатывает задачу по управлению относительно одноразовыми экземплярами Pod. Таким образом, хоть можно напрямую использовать Pod, в Kubernetes гораздо более удобно управлять своими Pod с помощью контроллера.

Контроллер может создавать и управлять несколькими Pod, обрабатывать репликацию, развертывание и предоставлять возможности

самовосстановления в области кластера. Например, если узел аварийный, контроллер может автоматически заменить Pod, запланировав идентичную замену на другом узле.

Некоторые примеры контроллеров, которые содержат один или несколько модулей, включают:

- Deployment;
- StatefulSet;
- DaemonSet.

В общем, контроллеры используют шаблон Pod-ов, для их создания, и несет за них ответственность.

В шаблонах Pod описываются спецификации pod-ов, которые включены в другие объекты, такие как контроллеры репликации, Jobs и DaemonSets. Контроллеры используют шаблоны Pod для создания реальных контейнеров. Пример ниже - простой манифест для Pod, который содержит контейнер, который печатает сообщение.

```
apiVersion:v1
kind:Pod
metadata:
name: kymbatapp-pod
labels:
app: kymbatapp
spec:
containers:
-name: kymbatapp-container
image: Aues-image
command:['sh','-c','echo Hello AUES! && sleep 3600']
```

Вместо того, чтобы указывать текущее желаемое состояние всех реплик, шаблоны пакетов выглядят как yaml отрезки. Нет сложного представления. Последующие изменения в шаблоне или даже переход на новый шаблон не оказывают прямого влияния на уже созданные контейнеры. Аналогично, контейнеры, созданные контроллером репликации, впоследствии могут быть напрямую обновлены. Это преднамеренно контрастирует с Pod, которые определяют текущее желаемое состояние всех контейнеров, принадлежащих Pod. Этот подход радикально упрощает семантику системы и повышает гибкость примитива.

1.2 Службы

В Kubernetes Pod-ы похожи на процессы в ОС, они появляються, и когда завершаються, они не воссоздаються как предыдущая версия. ReplicaSets, в частности, создают и уничтожают Pod динамически (например, при масштабировании вверх или вниз). Хоть и Pod получает уникальный IP адрес на время пока работает данные Pod, и лишаеться по завершению своей

работы. То есть данная сущьность не являеться статической и в рамках кластера приходиться реализовывать иное решение от стандартного локального интерфейса взаимодействия контейнеров внутри одного Pod. Так как Pod-ы не могут быть разбиты по нескольким узлам кластера.

Служба Kubernetes - это решение создающий логическую прослойку между Pod и кластером, группируемый по определенным признакам. Набор пунктов, предназначенных для службы, определяется (обычно) с помощью селектора меток.

В качестве примера рассмотрим бэкенд обработки изображений, который работает с 3 репликами. Эти реплики являются взаимозаменяемыми - интерфейсы не заботятся о том, какие бэкэнд они используют. В то время как фактические Pods, которые составляют набор бэкэндов, могут измениться, клиентские интерфейсы не должны знать об этом или отслеживать список самих бэкэндов. Абстракция службы позволяет эту развязку.

Для собственных приложений Kubernetes предлагает простой API конечных точек, который обновляется всякий раз, когда изменяется набор Pod в службе. Для не-родных приложений Kubernetes предлагает виртуальный IP-мост для служб, которые перенаправляются на бэкэнд-каналы.

Служба в Kubernetes - объект REST, подобен Pod-у. Как и все объекты REST, для создания нового экземпляра приложение POST может быть отправлено POST запросом. Например, предположим, что есть набор Pod, каждый из которых слушает порт 1234 и несет метку «app = KymbatApp».

kind:Service
apiVersion:v1
metadata:
name: kymbat-service
spec:
selector:
app: Kymbat*App*
ports:
-protocol:TCP
port:80
targetPort:1234

Эта спецификация создаст новый объект службы с именем «my-service», который предназначен для TCP-порта 1234 на любом Pod с меткой «app = KymbatApp». Этой службе также будет назначен IP-адрес (иногда называемый «IP-адрес кластера»), который используется прокси-серверами службы (см. Ниже). Селектор службы будет оцениваться непрерывно, и результаты будут отправлены в объект Endpoints, также называемый «my-service».

Обратите внимание: служба может отображать входящий порт в любой целевой порт. По умолчанию targetPort будет установлен на то же значение, что и поле порта. Возможно, более интересным является то, что targetPort

может быть строкой, ссылаясь на имя порта в backend Pod. Фактический номер порта, присвоенный этому имени, может быть разным в каждом бэкэнд-Pod. Это обеспечивает большую гибкость при развертывании и развитии услуг. Например, можно изменить номер порта, который слушает pod в следующей версии программного обеспечения, без разрыва клиентов.

Службы Kubernetes поддерживают протоколы TCP, UDP и SCTP. По умолчанию используется TCP.

Службы, как правило, абстрактного доступа к Kubernetes Pod, но они также могут абстрагироваться от других видов бэкэндов. Например:

- Нужно создать внешний кластер базы данных, но в тесте используеться свои собственные базы данных;
- Нужно указать службу на службу в другом пространстве имен или на другом кластере;
- Нужно перенести рабочую нагрузку на Kubernetes, а некоторые из серверов работают за пределами Kubernetes.

В любом из этих сценариев вы можете определить сервис без селектора:

```
kind:Service
apiVersion:v1
metadata:
name: kymbat-service
spec:
ports:
-protocol:TCP
port:80
targetPort:1234
```

Поскольку у этой службы нет селектора, соответствующий объект Endpoints не будет создан. Можно вручную сопоставить службу с конечными точками:

```
kind:Endpoints
apiVersion:v1
metadata:
name: kymbat-service
subsets:
-addresses:
-ip:1.2.3.4
ports:
-port:1234
```

Доступ к службе без селектора работает так же, как если бы у него был селектор. Трафик будет маршрутизирован в конечные точки, определенные пользователем (1.2.3.4:1234 в этом примере).

Служба ExternalName является особым случаем службы, которая не имеет селекторов и вместо этого использует DNS-имена.

Каждый узел в кластере Kubernetes запускает kube-proxy. kube-proxy отвечает за реализацию формы виртуального IP для служб типа, отличных от ExternalName.

В Kubernetes v1.0 Services являются конструкцией L4 (TCP / UDP over IP), прокси-сервер был исключительно в пользовательском пространстве. В Kubernetes v1.1 был добавлен API-интерфейс Ingress (бета) для представления услуг уровня «1» (HTTP), прокси-сервер iptables также стал стандартным режимом работы с Kubernetes v1.2. В Kubernetes v1.8.0-beta.0 добавлен ipvs-прокси.

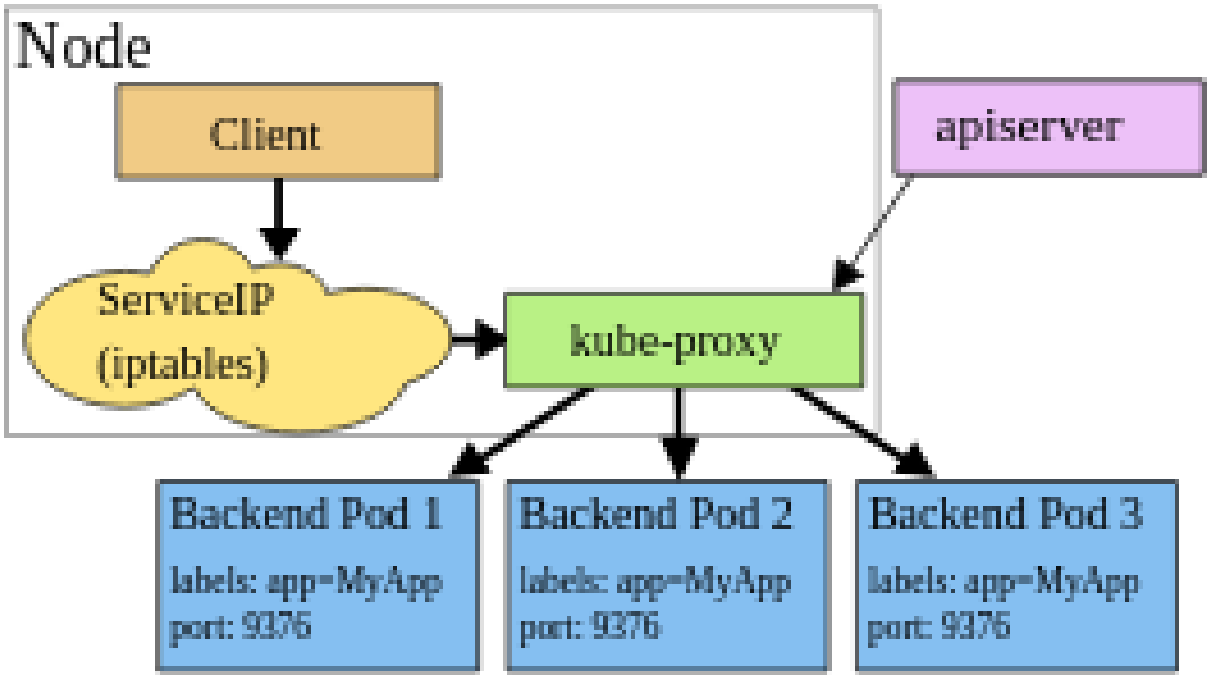

Рисунок 1.3 - Балансировка в Kubernetes на прокси

Вы можете указать свой собственный IP-адрес кластера как часть запроса на создание службы. Для этого установите поле .spec.clusterIP. Например, если уже есть существующая запись DNS, которую нужно повторно использовать, или устаревшие системы, настроенные для определенные IP-адреса, и их трудно переконфигурировать. IP-адрес, который пользователь выбирает, должен быть действительным IP-адресом и диапазоном CIDR для службы-кластера-ip-диапазона, который указан флагом на сервере API. Если значение IP-адреса недопустимо, apiserver возвращает код состояния 422 HTTP, чтобы указать, что это значение недопустимо.

1.3 Безопасность

Объекты “Secret” типа предназначены для хранения конфиденциальной информации, такой как пароли, токены OAuth и ключи ssh. Предоставление этой информации в секрет является более безопасным и более гибким, чем дословное вложение в определение контейнера или образа Docker. Пользователи и система могут создавать секреты. Чтобы использовать секрет, Pod должен ссылаться на секрет. Секрет может использоваться с модулем двумя способами: как файлы в томе, установленном на одном или нескольких его контейнерах, или используемом kubelet при скачивании образов для Pod.

ServiceAccounts автоматически создают и присоединяют секреты с учетными данными API. Kubernetes автоматически создает секреты, которые содержат учетные данные для доступа к API, и автоматически изменяет контейнеры, чтобы использовать этот тип секретных данных. Автоматическое создание и использование учетных данных API можно отключить или переопределить, если это необходимо. Однако, если все, что нужно сделать, это безопасный доступ к apirusver, то это рекомендуемый рабочий процесс. Квота ресурсов, определенная объектом ResourceQuota, предоставляет ограничения, которые ограничивают совокупное потребление ресурсов на пространство имен. Он может ограничить количество объектов, которые могут быть созданы в пространстве имен по типу, а также общий объем вычислительных ресурсов, которые могут потребляться ресурсами в этом проекте.

Квоты ресурсов работают следующим образом:

- различные команды работают в разных пространствах имен. В настоящее время это является добровольным, но планируется поддержка этого обязательного с помощью ACL;
- администратор создает один или несколько ResourceQuotas для каждого пространства имен;
- пользователи создают ресурсы (контейнеры, службы и т. д.) В пространстве имен, а система квот отслеживает использование, чтобы гарантировать, что она не превышает ограничения жестких ресурсов, определенные в ResourceQuota;
- если создание или обновление ресурса нарушает ограничение квоты, запрос будет с кодом HTTP 403 FORBIDDEN с сообщением, объясняющим ограничение, которое было бы нарушено;
- если квота включена в пространстве имен для вычислительных ресурсов, таких как процессор и память, пользователи должны указывать запросы или лимиты для этих значений; в противном случае система квот может отклонить создание пакета. Подсказка: используйте контроллер доступа LimitRanger, чтобы принудительно устанавливать значения по умолчанию для контейнеров, которые не требуют вычислительных ресурсов.

Примеры политик, которые могут быть созданы с использованием пространств имен и квот:

- в кластере с емкостью 32 гигабайта и 16 ядрах, пусть команда A использует 20 гигабайт и 10 ядер, пусть B использует 10GiB и 4 ядра, и держит 2GiB и 2 ядра в резерве для будущего распределения;

- ограничьте пространство имен «test», используя 1 ядро и 1GiB RAM. Пусть пространство имен «production» использует любое значение.

В случае, когда общая емкость кластера меньше суммы квот пространств имен, может существовать конкуренция за ресурсы. Это обрабатывается по принципу «первым-пришел первым-вышел». Ни одно утверждение, ни изменения квоты не повлияют на уже созданные ресурсы.

Политика безопасности Pod - это ресурс на уровне кластера, который контролирует чувствительные к безопасности аспекты спецификации pod. Объекты PodSecurityPolicy определяют набор условий, которые должен выполняться модулем для приема в систему, а также значения по умолчанию для связанных полей. Управление политикой безопасности Pod осуществляется как дополнительный (но рекомендуемый) контроллер допуска. PodSecurityPolicies обеспечивается принудительным включением контроллера допуска, но при этом без авторизации каких-либо политик будет предотвращено создание в кластере каких-либо модулей.

Поскольку API политики безопасности pod (policy / v1beta1 / podsecuritypolicy) включен независимо от контроллера допуска, для существующих кластеров рекомендуется, чтобы политики были добавлены и авторизованы до включения контроллера доступа. Когда создается ресурс PodSecurityPolicy, он ничего не делает. Чтобы использовать его, учетная запись запрашивающего пользователя или целевого контейнера должна иметь право использовать эту политику, разрешая использовать глагол использования политики.

Большинство модулей Kubernetes создаются не непосредственно пользователями. Вместо этого они обычно создаются опосредованно как часть диспетчера развертывания, ReplicaSet или другого шаблонного контроллера через диспетчер контроллера. Предоставление доступа контроллеру к политике обеспечит доступ ко всем модулям, созданным этим контроллером, поэтому предпочтительный метод авторизации политик - предоставить доступ к учетной записи службы pod.

1.4 Журналирование

Журналы приложений и систем могут помочь понять, что происходит внутри кластера. Журналы особенно полезны для отладки проблем и мониторинга активности кластера. Большинство современных приложений имеют какой-то механизм журналов; как таковые, большинство контейнерных систем также разработаны для поддержки какого-то ведения журнала. Самый простой и наиболее распространенный метод ведения журналов для контейнеризованных приложений - это запись в стандартный вывод и стандартные потоки ошибок.

Тем не менее, нативная функциональность, предоставляемая движком контейнера или временем выполнения, обычно недостаточно для полного решения журналирования. Например, если контейнер выходит из строя, выгружается модуль или умирает узел, обычно необходимо получить доступ к журналам приложения. Таким образом, журналы должны иметь отдельное хранилище и жизненный цикл, не зависящие от узлов, или контейнеров. Эта концепция называется кластерным протоколом. Для ведения журналов на уровне кластеров требуется отдельный бэкэнд для хранения, анализа и запросов журналов. Kubernetes не предоставляет встроенное решение для хранения данных журнала, но можно интегрировать многие существующие решения для ведения журналов в кластер Kubernetes.

Все контейнерное приложение записывает в stdout, а stderr обрабатывается и перенаправляется где-то с помощью контейнера. Например, механизм контейнера Docker перенаправляет эти два потока на драйвер ведения журнала, который настроен в Kubernetes для записи в файл в формате json. Драйвер регистрации Docker, json рассматривает каждую строку как отдельное сообщение. При использовании драйвера журнала Docker нет прямой поддержки многострочных сообщений. Необходимо обрабатывать многострочные сообщения на уровне агента регистрации или выше.

По умолчанию, если контейнер перезапускается, kubelet сохраняет один завершенный контейнер со своими журналами. Если из узла выдворяется Pod, все соответствующие контейнеры также высылаются вместе с их журналами. Важное соображение в протоколе уровня узла - это внедрение оборота журнала, так что журналы не потребляют все доступное хранилище на узле. В настоящее время Kubernetes не несет ответственности за ротацию журналов, но инструмент развертывания должен установить порог для решения этой проблемы. Например, в кластерах Kubernetes, развернутых сценарием kube-up.sh, есть инструмент логротата, настроенный для запуска каждый час. Также можно настроить время выполнения контейнера для автоматического ротации журналов приложения, например. используя log-opt Docker. В сценарии kube-up.sh последний подход используется для образов COS, и прежний подход используется в любой другой среде. В обоих случаях, по умолчанию, настройка ротации выполняется, когда файл журнала превышает 10 МБ.

В то время как Kubernetes не обеспечивает собственное решение для регистрации на уровне кластера, существует несколько общих подходов. Вот несколько вариантов:

- используйте агент регистрации на уровне узла, который выполняется на каждом узле;
- включите специальный контейнер журналов для регистрации в приложении;
- принимайте журналы непосредственно на бэкэнд из приложения.

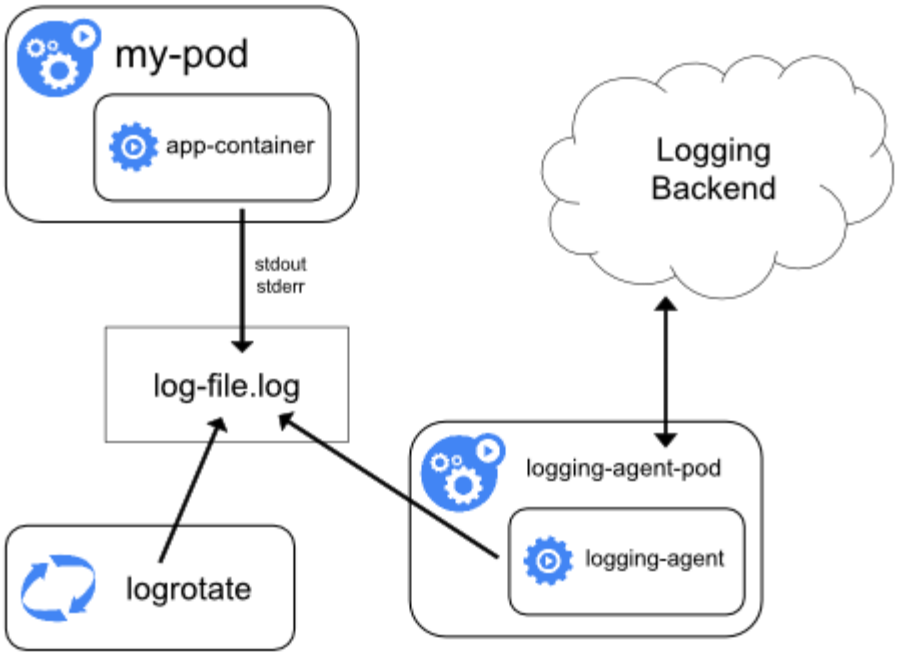

Рисунок 1.4 - Система логирования

Можно реализовать ведение журнала на уровне кластера, включив агент регистрации на уровне узла на каждой ноде. Агент регистрации является специальным инструментом, который предоставляет журналы или посылает журналы на бэкэнд. Как правило, агент регистрации является контейнером, который имеет доступ к каталогу с файлами журнала из всех контейнеров приложений на этом узле. Поскольку агент регистрации должен работать на каждом узле, его обычно реализуют как реплику DaemonSet, манифестный модуль , или выделенный собственный процесс на узле. Однако последние два подхода являются устаревшими и сильно обескуражены. Использование агента регистрации на уровне узла является наиболее распространенным и рекомендуемым подходом для кластера Kubernetes, поскольку он создает только один агент на узел и не требует каких-либо изменений в приложениях работающий на узле. Тем не менее, ведение журнала на уровне узлов работает только для стандартного вывода приложений и стандартной ошибки. Kubernetes не указывают агента регистрации, но два дополнительных агента регистрации собираются с выпуском Kubernetes: Stackdriver Logging для использования с облачной платформой Google и Elasticsearch. Оба используют fluentd с настраиваемой конфигурацией в качестве агента на узле.

Как упоминалось выше, fluentd, используемый daemonset, знает, как обрабатывать исключения для различных приложений, но fluentd чрезвычайно гибкий и может быть настроен так, чтобы разбивать сообщения журнала и модифицировать в зависимости от типа собираемых журналов. Также следует отметить, что fluentd добавляет полезные метаданные Kubernetes в журналы, которые могут пригодиться в более крупных средах, состоящих из нескольких

узлов и контейнеров. Ниже приведены несколько примеров того, как можно использовать эти метаданные для получения видимости в кластере Kubernetes с визуализацией Kibana. Матричные визуализации просты и отлично подходят для отображения простой статистики, связанной с настройкой. Например, можно использовать уникальную агрегирование count поля kubernetes.container_name, чтобы узнать, сколько контейнеров используется в каждом pod.

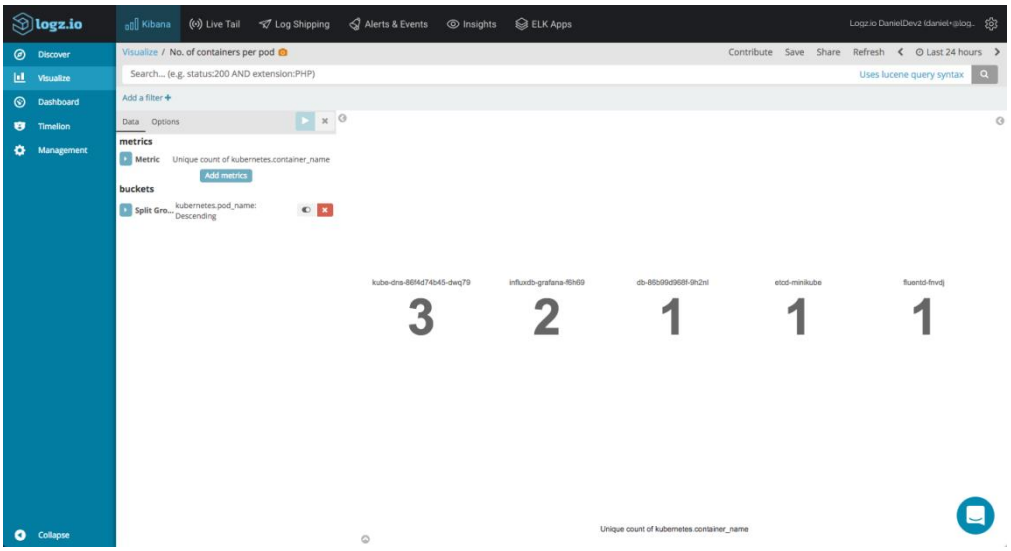

Рисунок 1.5 - Главная консоль Kibana

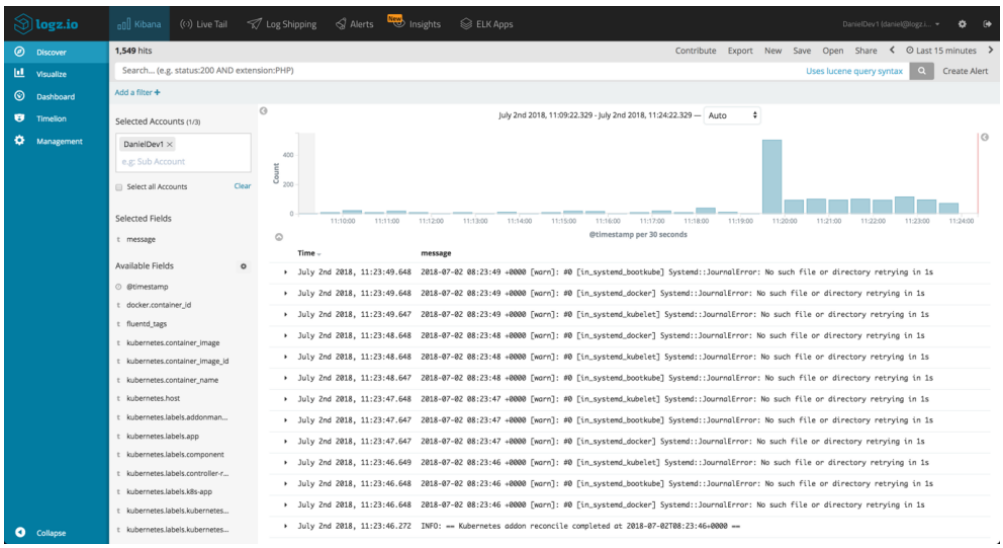

Рисунок 1.6 - Журналы в Kibana

1.5 Мониторинг

Чтобы масштабировать приложение и предоставлять надежный сервис, нужно понять, как работает приложение при его развертывании. Проверить производительность приложения в кластере Kubernetes, изучив контейнеры,

службы и характеристики общего кластера. Kubernetes предоставляет подробную информацию об использовании ресурсов приложения на каждом из этих уровней. Эта информация позволяет оценивать производительность приложения и устранять узкие места, чтобы повысить общую производительность.

- Resourcemetricspipeline;
- Fullmetricspipelines;
- CronJobmonitoring

В Kubernetes мониторинг приложений не зависит от одного решения для мониторинга. В новых кластерах можно использовать два отдельных конвейера для сбора статистики мониторинга по умолчанию:

Конвейер метрик ресурсов предоставляет ограниченный набор показателей, относящихся к компонентам кластера, таким как контроллер HorizontalPodAutoscaler, а также утилиту kubectl top. Эти показатели собираются метрикой-сервером и отображаются через API metrics.k8s.io. metrics-server обнаруживает все узлы кластера и запрашивает Kubelet каждого узла для использования ЦП и памяти. Kubelet извлекает данные из cAdvisor. metrics-server - это легкая кратковременное хранилище.

Полный стек метрик, такой как Prometheus, дает доступ к более богатым метрикам. Кроме того, Kubernetes может реагировать на эти показатели, автоматически масштабируя или адаптируя кластер на основе его текущего состояния, используя такие механизмы, как Horizontal Pod Autoscaler. Протокол мониторинга извлекает метрики из Kubelet, а затем предоставляет их Kubernetes через адаптер, реализуя либо API custom.metrics.k8s.io, либо external.metrics.k8s.io.

Оператор Prometheus упрощает настройку Prometheus на Kubernetes и позволяет обслуживать API пользовательских показателей с помощью адаптера Prometheus. Prometheus обеспечивает надежный язык запросов и встроенную панель мониторинга для запросов и визуализации данных. Полный «контроль Kubernertes со стеком Prometheus» состоит из гораздо большего, чем серверов Prometheus, которые собирают метрики, очищая конечные точки. Чтобы развернуть реальное решение для мониторинга Kubernetes и микросервисов, много других поддерживающих компонентов, включая правила и предупреждения (AlertManager), уровень графической визуализации (Grafana), хранилище долгосрочных метрик, а также дополнительные адаптеры показателей для программного обеспечения, которое несовместимо из коробки.

Проект Grafana является агностической платформой для анализа и мониторинга. Он не связан с Prometheus, но стал одним из самых популярных добавочных компонентов для создания полного решения Prometheus.

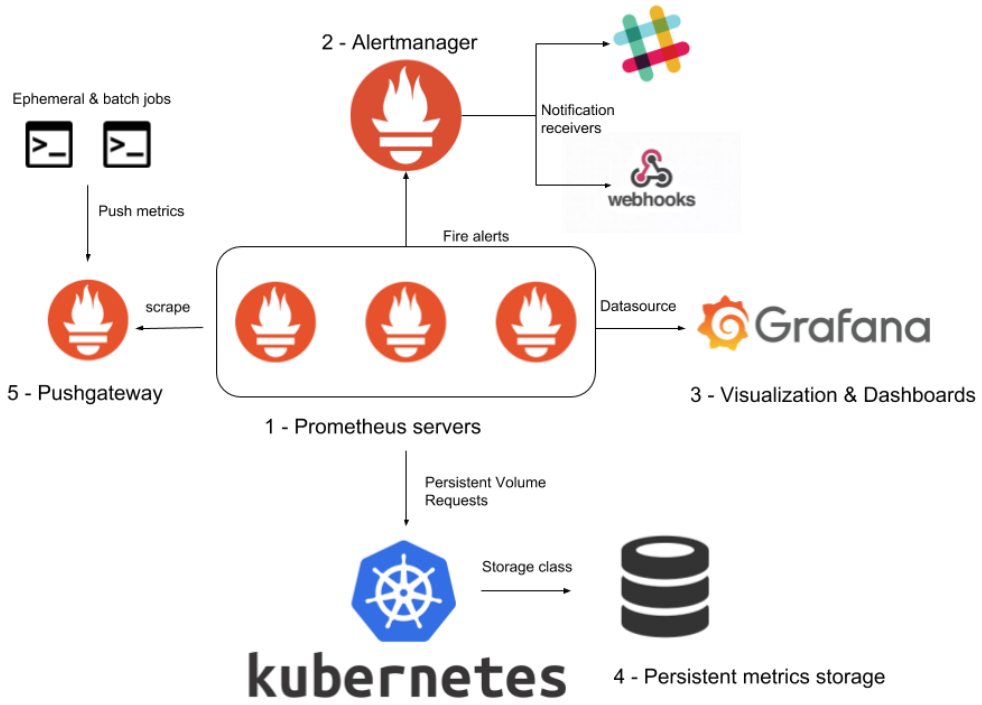

Рисунок 1.7 - Система мониторинга Prometheus

Рисунок 1.8 – Grafana

1.6 Service mesh

Service mesh представляет собой класс решения микросервисной архитектуры для инфраструктуры современных приложений. Хоть и присутствие Istio на рынке относительно недавно, но этот продукт стал быстро популярен в широком кругу специалистов, комплексность в плане

предоставляемых функционала, требует от специалиста большого времени для знакомства.

Созданный в сотрудничестве с крупными ИТ гигантами в отрасли(IBM, Google) проект с открытым кодом – Istio, решает проблемы встречаемые в приложениях микросервисной архитектуры, например как:

а) контроль трафика: таймауты, переотправка запросов, распределение нагрузки;

б) безопасность: верификация пользователей, шифрование, сокрытие источника;

в) мониторинг: трекинг запросов, диагностика, журналирование.

Вышеописанные проблемы могут быть решены на уровне приложения, но после такого вмешательства трудно будет называть сервисы “микро”. И методы нацеленные для решения этих преград потребляют единицу ресурсов которые могли бы быть использованы для нужно бизнес задачь. Рассмотрим пример:

Необходимо определить время для реализации функции обратной связи. Для чего требуеться CRUD, верифицикация пользователей и сервисов, переотправка запросов, алгоритм обрыва цепи, настройка таймаута запросов, мониторинг для отслеживания метрик сервиса и система алертинга при превышении пороговых значений, трекинг запросов, и т.д. Объём усилий, требующихся для добавления единичного сервиса, огромен. Как же Istio удаеться устранять описанные выше проблемы, не относящимся к задачам бизнеса.

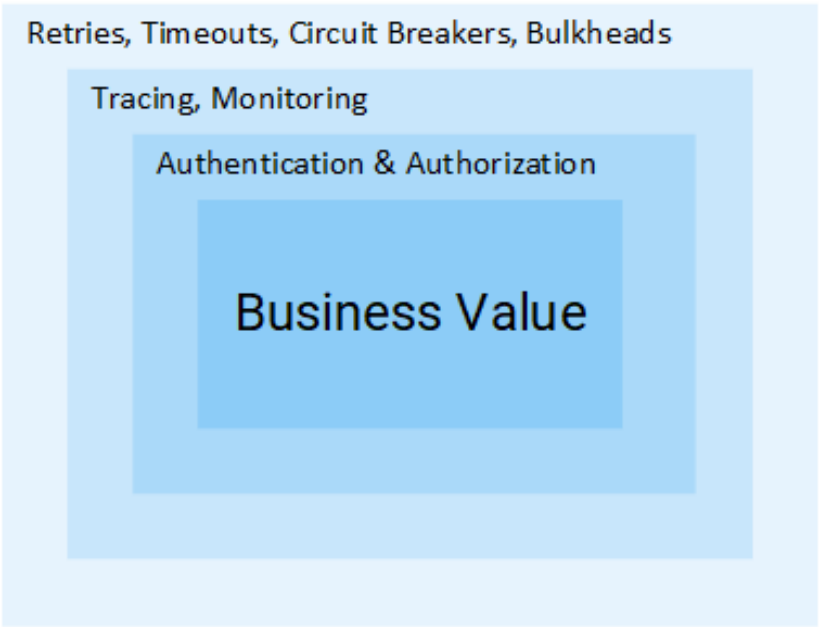

Рисунок 1.9 Инкапсуляция Бизнес задачи

2 Istio

В классической архитектуре сервисы обмениваются запросами напрямую, в случае превышения порогового времени ожидания ответа, сервис берет на себя всю обработку этого запроса.

Рисунок 2.1 - Сетевой трафик в Kubernetes

В архитектуре Istio же присутствует посредник между сервисами(для каждого сервиса свой посредник), то есть по сути та же классическая схема но в качестве взаимодействующих узлов выступают посредники, такой подход реализует:

- надежность: по полученному статусу запроса, посредники решают каким путем направить запрос по определенному алгоритму;
- канареечное развертывание: балансирование нагрузки с определенным весом на выборочное число конечных сервисов;
- мониторинг: статус сервисов, сбор метрик;
- трекинг: отслеживание запроса по заголовку внутри Kubernetes кластера;
- безопасность: извлекает JWT-токен, верифицирует пользователей.

2.1 Архитектура Istio

Посредник в виде второго контейнера в pod-е выступает в роли прокси сервиса основному контейнеру, и пропускает весь трафик через себя применяет к нему набор правил. Посредники, имеющие весь функционал, образуют Data Plane, и становятся доступными для настройки через Control Plane.

Data Plane: контейнер посредника отвечающий некоторым базовым требованиям по отношению к основному контейнеру сервиса, например, функции переотправки запросов.

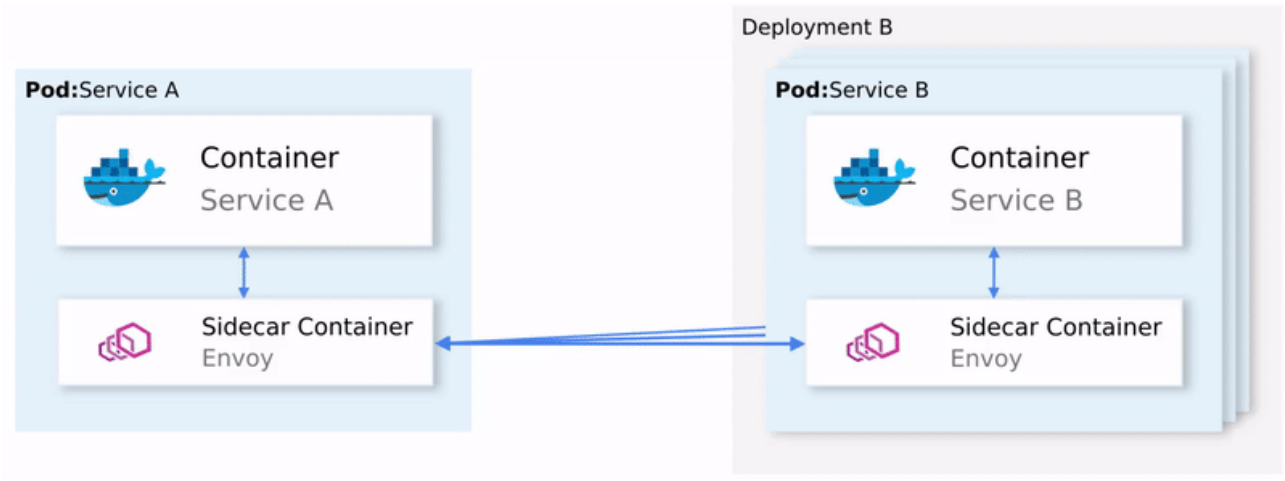

Рисунок 2.2 - Реализация retries и circuit breaking в Envoy

а) Envoy прокси*(посредник, вторичный контейнер)* отправляет запрос основному контейнеру сервиса B где возникает сбой;

б) Envoy посредник пытается переотправить запрос;

в) Посредник получает ответ о сбое;

г) Включается механизм обрыва цепи и посредник береться за обработку последующих запросов.

С собственной реализацией возможности обвесных сервисов по сравнению с вложенным функционалом предоставляемый Istio, то заметно что задача займет не мало времени. А у Istio оно уже имеется, и готово к работе практически сразу.

Control Plane: включает в себя три основных компонента: Citadel, Pilot и Mixer в комбинации строят маршрутизацию для сервисов основного контейнера. А так же выполняют дополнительные операции по управлению трафиком и сбора метрик:

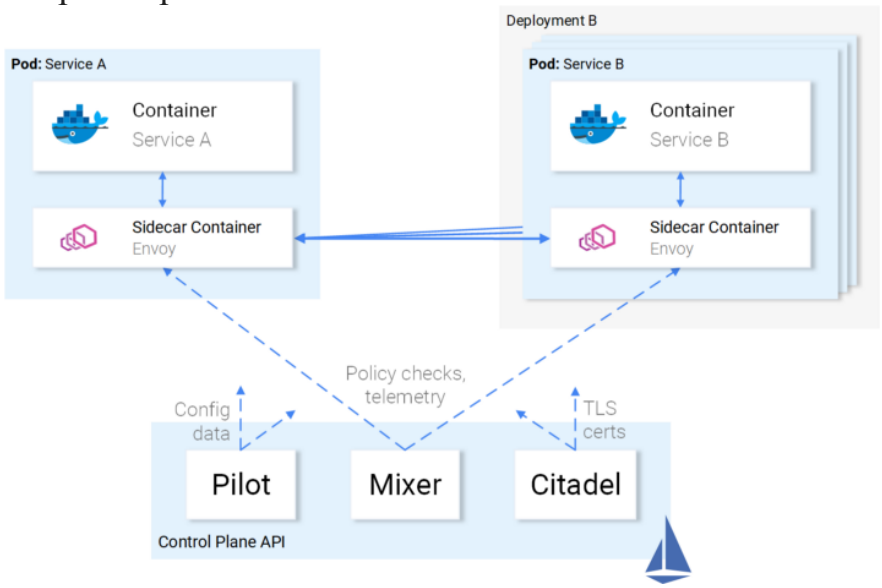

Рисунок 2.3 - Взаимодействие Control Plane с Data Plane

Data Plane представляеться в виде CRD(Custom Resource Definitions) манифеста для Kubernetes кластера. То есть, это такой же ресурс в Kubernetes со знакомым синтаксисом, который легко править по необходимости. При применении манифеста Control Plane заметит этот ресурс и возьмет под контроль Data Plane.

2.2 Ingress Gateway

Для связи кластера Kubernetes с внешним миром нужен шлюз, и в случае Istio рекомендуемый вариант использовать Ingress Gateway. По сути это http/https прокси который можно реализовать несколькими вариантами. Самое распространенное решение – nginx, с функциями балансировки, шифрования, маршрутизации по содержимому, имени и таргету запроса. По умолчанию Istio блокирует(про попытке открыть страничку в браузере какого нибуть сервиса, выдает 500-ю ошибку HTTP протокола) все запросы пока не будет настроен шлюз.

Gateway так же описываеться с помошью CRD(Custom Resource Definition) в Kubernetes, и перед применение манифеста так же есть возможность изменить параметры порта, протокола, узла и т.д. Пример HTTP-трафика на 80-й порт для всех хостов. *kymbat-gateway.yaml*:

```
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: kymbat-gateway
spec:
  selector:
    istio: ingress-kymbat-gateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
- "*"
```

Конфигурация выше принимает все хосты по протоколу HTTP, 80 порту, имя шлюза – kymbat-gateway. Селектор *istio: ingress-kymbat-gateway* определяет к которому Ingress Gateway относиться эта конфигурация. В моем случае это шлюз установленный по умолчанию. На манифесте выше описан только один пример шлюза, и нет препятствий указать их большее число и разными наборами параметров.

Сейчас шлюзу извесно об источнике маршрутизируемых запросов, но нет информации о том куда они должны быть перенаправлены. Для этого

используеться Virtual Services. VirtualService определяет как маршрутизировать запросы внутрь кластера на Ingress Gateway.

Запросы к сервисам, поступющие через ingress контроллер, будут распределяться по путям как на рисунке ниже:

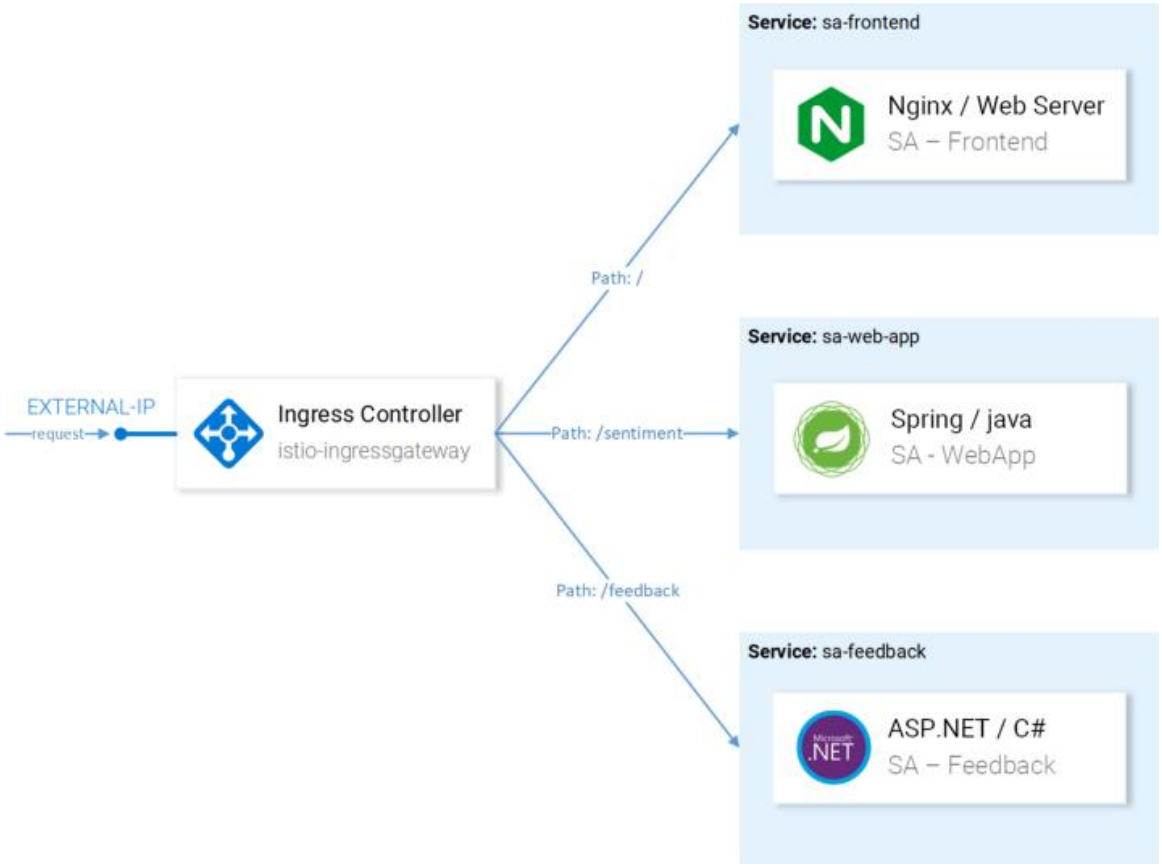

Рисунок 2.4 - Маршруты, которые необходимо настроить с VirtualServices

Запросы, направляемые на SA-Frontend:

- таргеты должны совпадать абсолютно полностью(/), при совпадении отправляеться в SA-Frontend;

- таргеты с префиксом /static/* пересылаються на SA-Frontend где получает в ответ CSS и JavaScript, для отображения статичного контента сервиса;

- таргеты, попадающие под выражение '^.*\.(png|jpg)$', будут отправлены на SA-Frontend, для отображения картинок в контенте ответа сервиса.

Конфигурация VirtualService приведена в файле ниже *kymbat-virtualservice-external.yaml:*

kind: VirtualService
metadata:

```
  name: kymbat-external-services
spec:
  hosts:
  - "*"
  gateways:
  - kymbat-gateway                    # 1
  http:
  - match:
    - uri:
        exact: /
    - uri:
        exact: /callback
    - uri:
        prefix: /static
    - uri:
        regex: '^.*\.(png|jpg)$'
    route:
    - destination:
        host: sa-frontend             # 2
        port:
number: 80
```

где:

- VirtualService связан со шлюзом *kymbat*-gateway, через который поступаю запросы;
- в destination пишется имя сервиса назначения запросов.

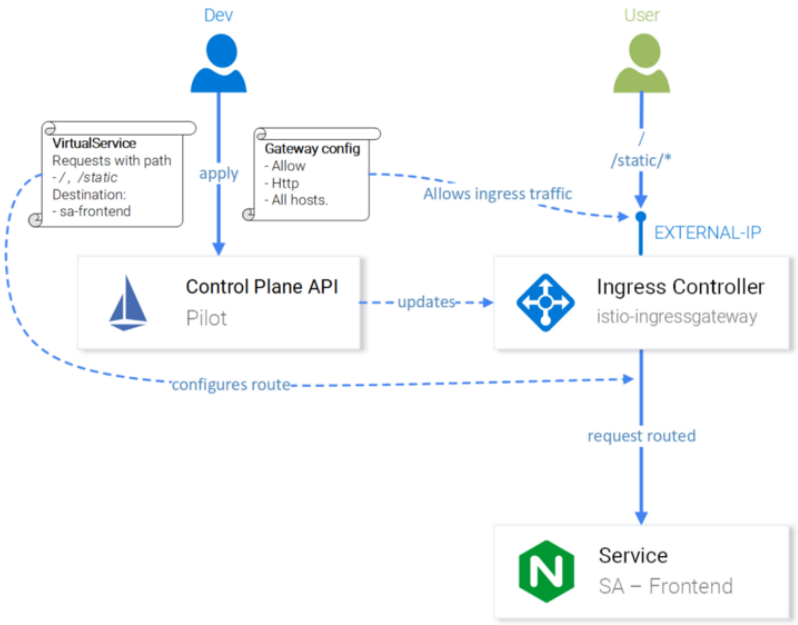

Рисунок 2.5 - Конфигурация Istio-IngressGateway для маршрутизации запросов

П р и м е ч а н и е – В файле *kymbat*-virtualservice-external.yaml содержиться настройки , в котором так же описаны конфигурации по маршрутизации по двум другим сервисам.

Сервис доступен по адресу прокси сервиса кластера http://{IP-Адресс-балансировщика}/.

2.3 Диагностика

Административный интерфейс Kiali (http://localhost:20001) Здесь болшой набор возможностей, как например, проверка конфигурации элементов Istio, графическое представление сервисов с сортировкой и интуитивным поиском, фильтрацией по заданным параметрам – источник/получатель, сбои при доставке и т.д.

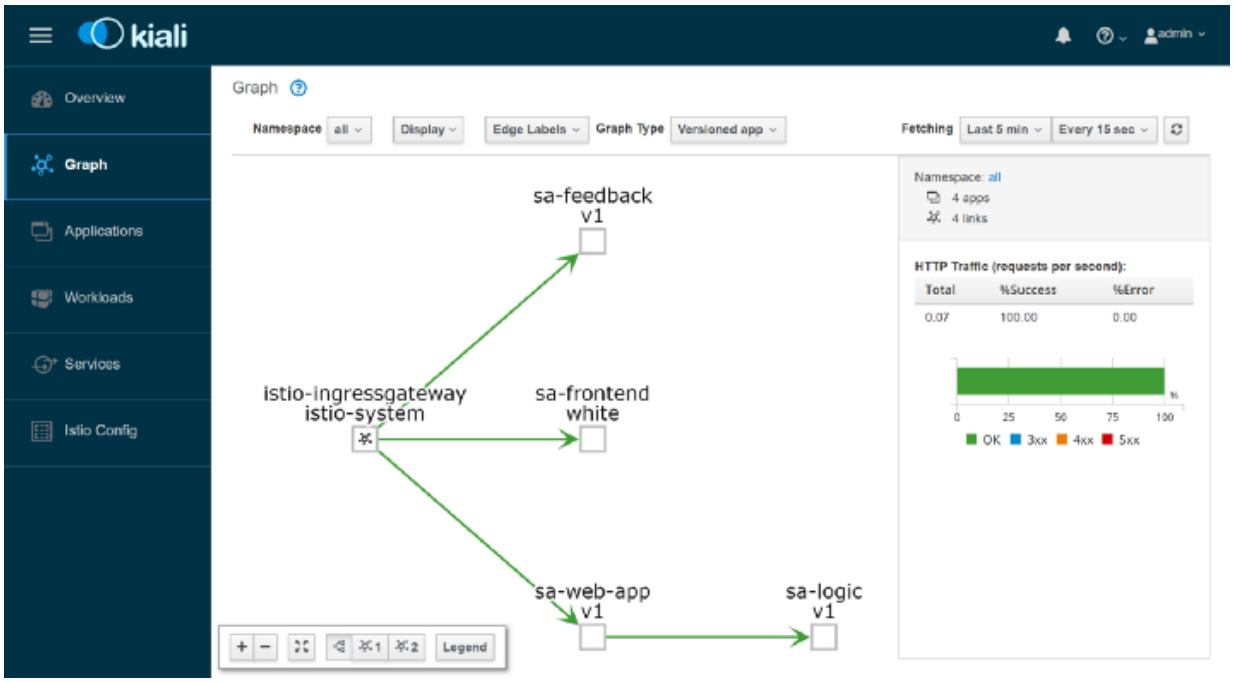

Рисунок 2.6 - Схема в Kiali

Собранные в Istio метрики попадают в Prometheus и визуализируются с Grafana. Административный интерфейс Grafana, (http://localhost:3000/). В левом верхнем углу меню Grafan-ы есть кнопка Istio Service Dashboard-а в выпадающий листе так же слева верхнем углу выберите сервис для просмотра метрик:

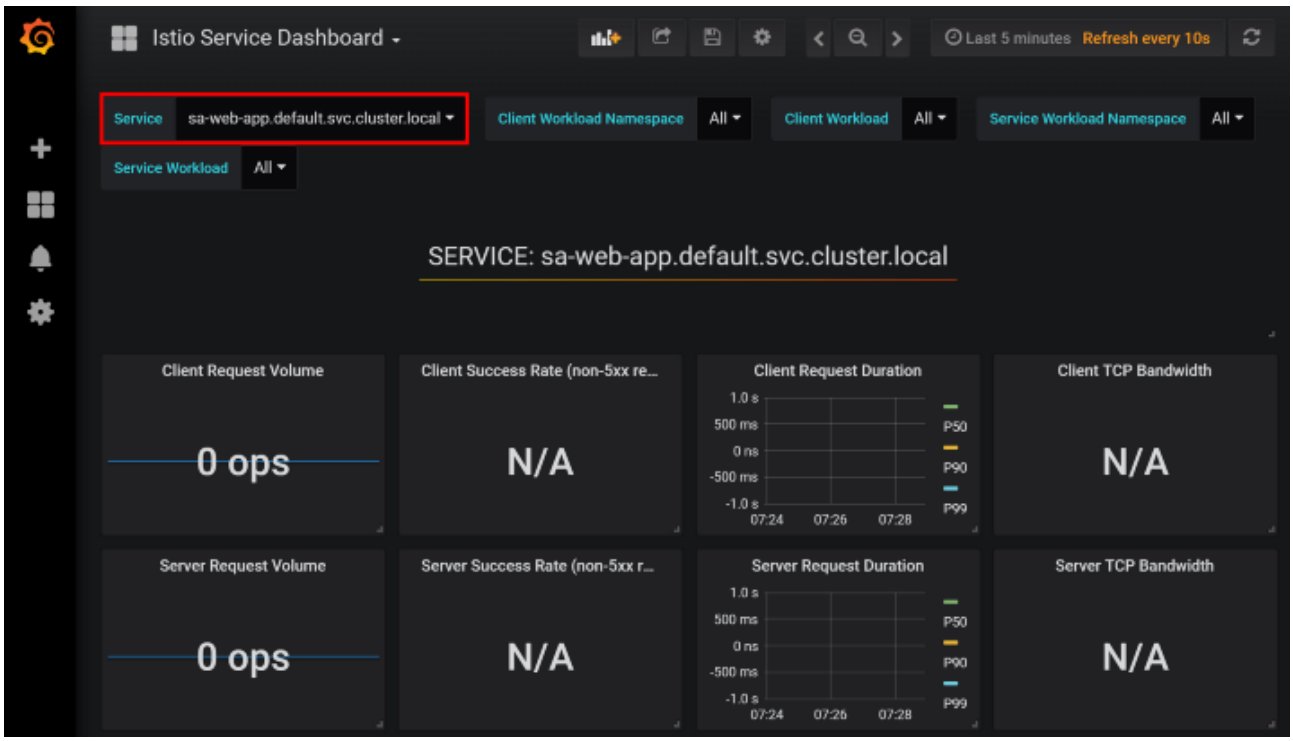

Рисунок 2.7 - Мониторинг сервиса

Трассировка трафика необходима для ускорения поиска причины проблемы, особенно это заменто при большом количестве сервисов:

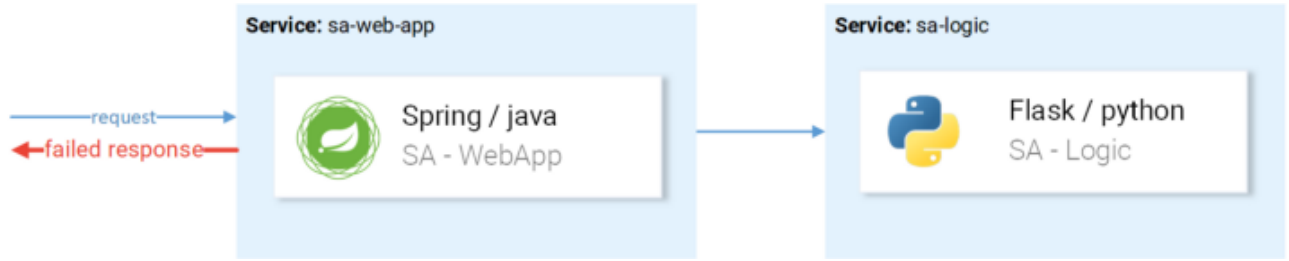

Рисунок 2.8 - Типовой пример случайного неудачного запроса

Запрос дошел до адресата и отправитель получает ответ об ошибке, для поиска причины воспользуемся трассировкой. На рисунке ниже виден пример типичной проблемы в обмене запросов между сервисами. Путем сравнения заголовка отправленного запроса с полученным заголовком на проблемном сервисе круг поиска проблемы сужаеться, сушествуют множество разных методик выявления проблем, но главным критерием являеться скорость нахождения и устранения проблемы:

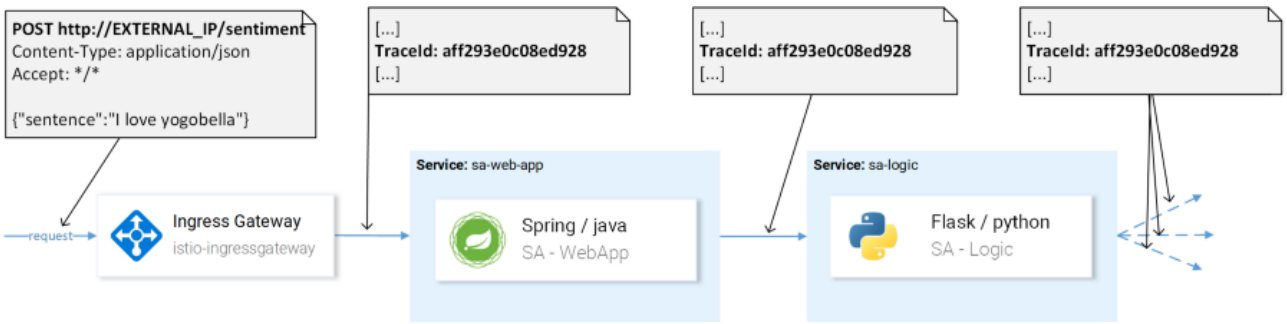

Рисунок 2.9 - Для идентификации запроса используется TraceId

Jaeger Tracer применяемый в Istio предоставляет фреймворк с API для трассировки трафика.

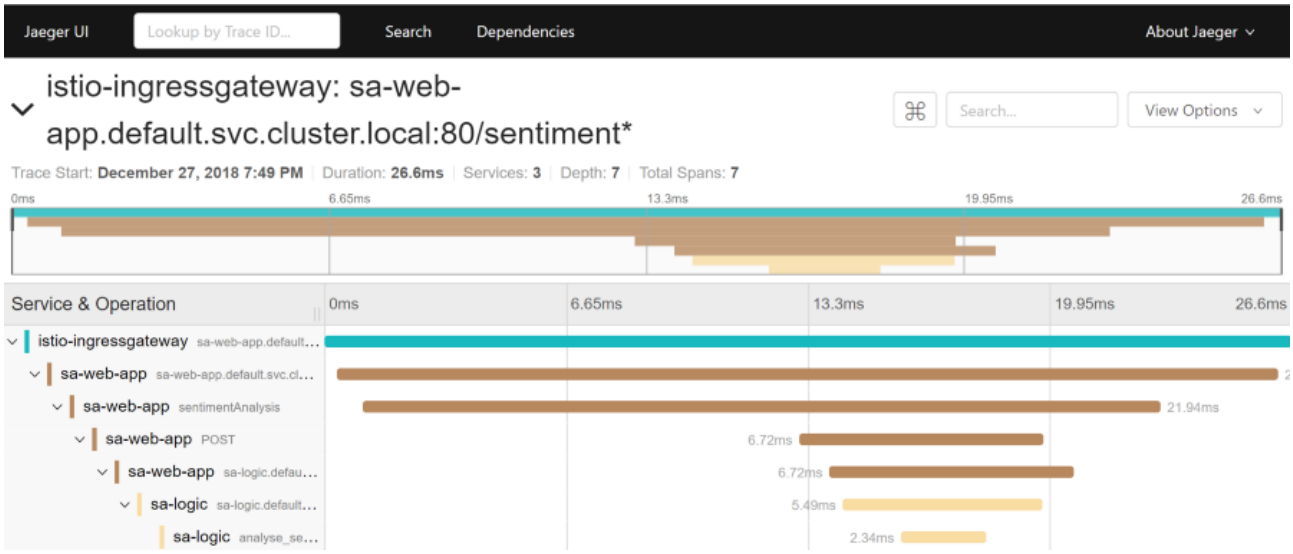

Рисунок 2.10 Трассировка запроса в Jaeger

Этот трейс показывает:

а) поступает запрос с меткой **istio-ingressgateway** (создаеться Trace Id, который будет отправной точкой для анализа), шлюз перенаправляет запрос на сервис **sa-web-app**;

б) поступивший в сервис **sa-web-app** запрос обрабатывается контейнером посредника, статус запроса отображаеться в span-е под Trace Id выше и перенаправляется в основной контейнер **sa-web-app**. Где span, определяет времея начала и продолжительность операции. В текущий трейс Span вложен иерархически последовательно по последовательности поступления или обработки запроса;

в) запрос обрабатывается **sentimentAnalysis**. Его трейс сгенерирован приложением;

г) генерируеться POST-запрос в **sa-logic**. Trace ID перекидываеться из **sa-web-app**.

П р и м е ч а н и е – Приложение получает заголовки генерируемые посредником, и продолжит направлять запросы по сценарию как на Рисунке 20:

Рисунок 2.11 - (A) За проброс заголовков отвечает Istio;
(B) За заголовки отвечают сервисы.

Istio выполняет большую часть по генерации заголовков входящих запросов, в каждом контейнере посредника создаются span-ы и перенаправляют их для отображения во время трассировки. Работа с пробросом заголовков критична так как если будет упущен хоть один заголовок то схема трассировки будет искажена. И важно помнить что существую заголовки забронированные системой, при перезатирании которых диагностика проблемы может запутать искателя.

Из заголовков запронированной Istio нужно включать в запросы следующее:

x-request-id
x-b3-traceid
x-b3-spanid
x-b3-parentspanid
x-b3-sampled
x-b3-flags
x-ot-span-context

Существует множество библиотек позволяющие выполнить эту не сложную операцию, упрощая управление настройками но увеличивая стек инструментов для работы с Istio. Например RestTemplate с зависимостями Istio выполняет проброску заголовков. Инженеру лищь остаетья настроить эту связку единыжды, из личного опыта необходимо всегда помнить о том что при долгом безуспешном анализе проблема так же может быть связана с заголовками на стороне посредника в моменте их обработки или проброски, самыми часто встречаемыми случаеми являеться:

1. Не пробрасываеться заголовок
2. Заголовок изменен и не опрабатываеться сервисом
3. Перезатирается заголовок

2.4 Управление трафиком

С Istio в кластере появляются новые возможности, позволяющие обеспечить:

-динамическую маршрутизацию запросов: канареечные выкаты, A/B-тестирование;

-балансировку нагрузки: простую и непротиворечивую, основанную на хэшах;

-восстановление после падений: таймауты, повторные попытки, circuit breakers;

-внесение неисправностей: задержки, обрыв запросов и т.п.

Эти возможности будут показаны на примере выбранного приложения и попутно представлены новые концепции. Первой такой концепцией станет DestinationRules *(т.е. правила о получателе трафика/запросов)*, с помощью которых мы активируем A/B-тестирование.

2.4.1 A/B-тестирование: DestinationRules на практике

A/B-тестирование применяется в случаях, когда существуют две версии приложения (обычно они отличаются визуально) и не уверены на 100%, какая из них улучшит взаимодействие с пользователем. Поэтому одновременно запускается обе версии и собираются метрики.

Для деплоя второй версии фронтенда, необходимой для демонстрации A/B-тестирования, выполняем следующую команду:

$ kubectl apply -f resource-manifests/kube/ab-testing/sa-frontend-green-deployment.yaml deployment.extensions/sa-frontend-green created

Манифест deployment'а для «зелёной версии» отличается в двух местах:

1) Образ основан на ином теге —istio-green;
2) Pod'ы имеют лейбл version: green

Поскольку оба deployment'а имеют лейбл app: sa-frontend, запросы, маршрутизируемые виртуальным сервисом sa-external-services на сервис sa-frontend, будут перенаправлены на все его экземпляры и нагрузка распределится посредством алгоритма round-robin, что приведёт к проблеме распределения одной сессии между балансируемыми нодами. В понятии Cisco – sticky session- это когда сессия устанавливается с балансируемой нодой, и все последующие пакеты по текушей сессии направляються на ту же ноду. Добиться этого просто с помощью непротиворечивой балансировки нагрузки на основе хэшей *(Consistent Hash Loadbalancing)*. В этом случае запросы от одного клиента отправляются в один и тот же экземпляр бэкенда, для чего используется предопределённое свойство — например, HTTP-заголовок. Реализуется с помощью DestinationRules.

Destination Rules.

После того, как **VirtualService** направил запрос в нужный сервис, с помощью DestinationRules можно определить политики, которые будут применяться к трафику, предназначаемому экземплярам этого сервиса:

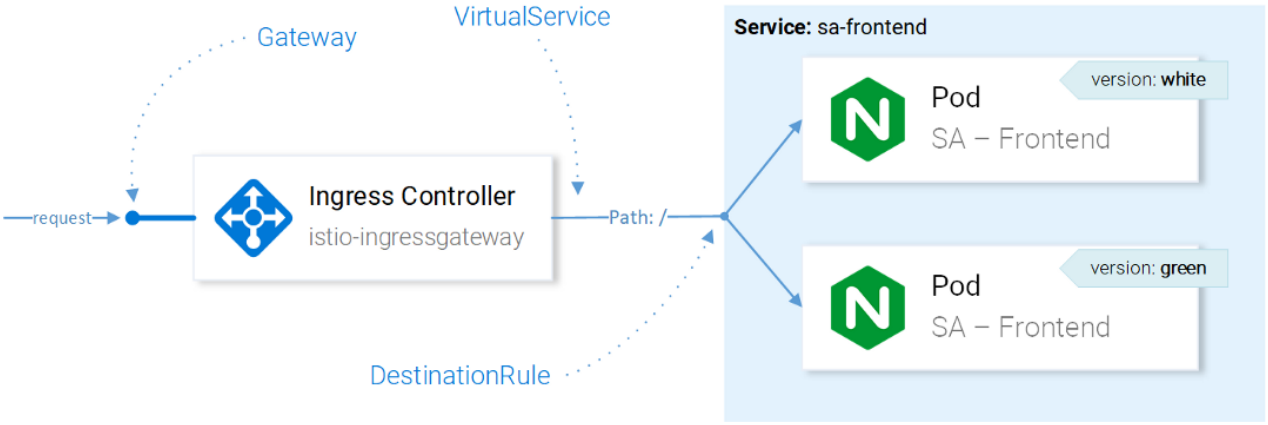

Рисунок 2.12- Управление трафиком с ресурсами Istio

Влияние ресурсов Istio на сетевой трафик представлено здесь в упрощённом для понимания виде. Если быть точным, то решение, на какой экземпляр отправлять запрос, делается Envoy'ем в Ingress Gateway, настроенным в CRD.

С помощью Destination Rules настроиваться балансировка нагрузки так, чтобы использовались непротиворечивые хэши и гарантировались ответы одного и того же экземпляра сервиса одному и тому же пользователю. Следующая конфигурация позволяет добиться этого:

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: sa-frontend
spec:
  host: sa-frontend
  trafficPolicy:
    loadBalancer:
      consistentHash:
        httpHeaderName: version
```

П р и м е ч а н и е — хэш будет генерироваться на основе содержимого HTTP-заголовка version.

2.4.2 Зеркалирование: Virtual Services на практике

Shadowing *(«экранирование»)* или Mirroring *(«зеркалирование»)* применяется в тех случаях, когда нужно протестировать изменение в production, не затронув конечных пользователей: для этого дублируется («зеркалируется») запросы на второй экземпляр, где произведены нужные изменения, и наблюдаются последствия. Чтобы проверить этот сценарий в действии, создадим второй экземпляр SA-Logic с багами (*nonfixed*).

$ kubectl apply -f sa-logic-service-nonfixed.yaml deployment.extensions/sa-logic-buggy created
$ kubectl get pods -l app=sa-logic --show-labels NAME READY LABELS sa-logic-s34rnv3i3b 2/2 app=sa-logic,version=v1 sa-logic-s34rn23r4v 2/2 app=sa-logic,version=v1 sa-logic-nonfixed-s34rnvsdv3w 2/2 app=sa-logic,version=v2 sa-logic-nonfixed-s34rnvsdHe9j 2/2 app=sa-logic,version=v2

Сервис sa-logic нацелен на pod'ы с лейблом app=sa-logic, поэтому все запросы будут распределены между всеми экземплярами:

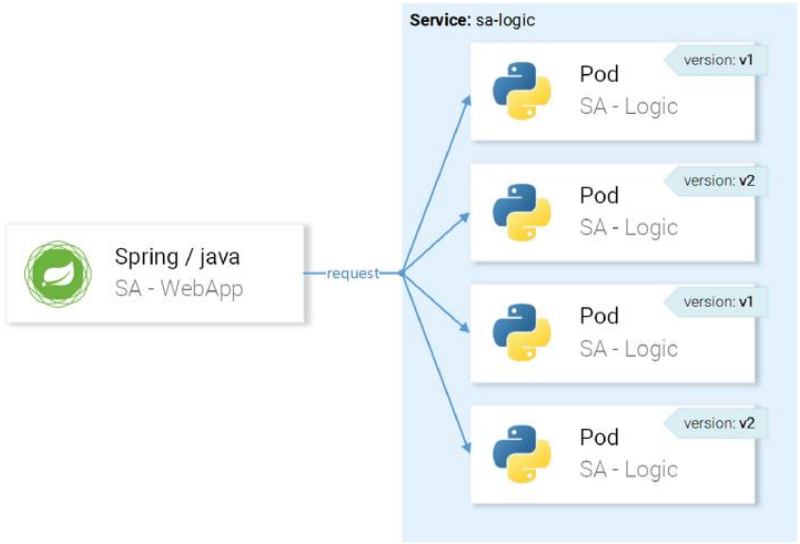

Рисунок 2.13- Маршрутизация по метке app=sa-logic

Для того, чтобы запросы направлялись на экземпляры с версией v1 и зеркалировались на экземпляры с версией v2, как это показано на Рисунке 23.

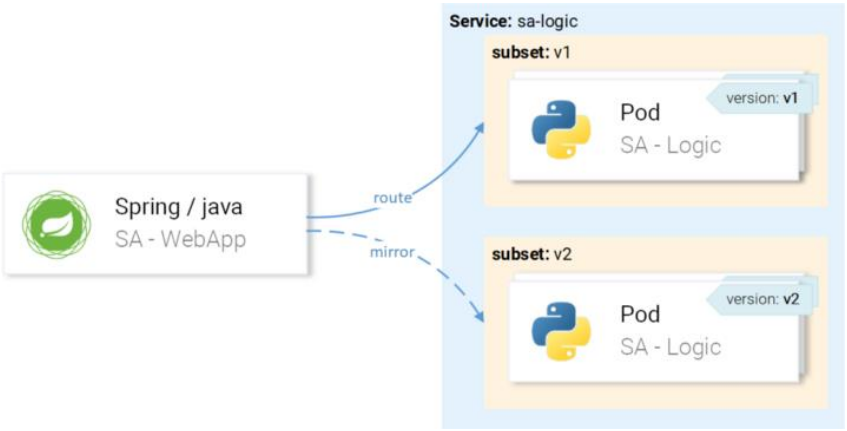

Рисунок 2.14- Маршрутизация по метке app=sa-logic и version=v1

Добиться этого можно через VirtualService в комбинации с DestinationRule, где правила определят подмножества и маршруты VirtualService к конкретному подмножеству.

Определение подмножеств в Destination Rules:

Подмножества *(subsets)* определяются следующей конфигурацией:

```
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: sa-logic
spec:
  host: sa-logic
  subsets:
  - name: v2
    labels:
      version: v2
  - name: v1
    labels:
      version: v1
```

П р и м е ч а н и я:

1 Хост (host) определяет, что это правило применяется только к случаям, когда маршрут идёт в сторону сервиса sa-logic

2 Названия (name) подмножеств используются при маршрутизации на экземпляры подмножества

3 Лейбл (label) определяет пары ключ-значение, которым должны соответствовать экземпляры, чтобы стать частью подмножества.

Когда подмножества определены, настройка VirtualService таким образом чтобы применить правила к запросам к sa-logic, чтобы они:

- маршрутизировались к подмножеству v1;
- зеркалировались к подмножеству v2.

```
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: sa-logic
spec:
  hosts:
    - sa-logic
  http:
  - route:
    mirror:
      host: sa-logic
      subset: v2
    - destination:
        host: sa-logic
        subset: v1
```

Создав искусственную нагрузку посмотрим на результаты в Grafana, где можно увидеть, что версия с багами (buggy) приводит к сбою для ~60 % запросов, но ни один из этих сбоев не затрагивает конечных пользователей, поскольку им отвечает работающий сервис.

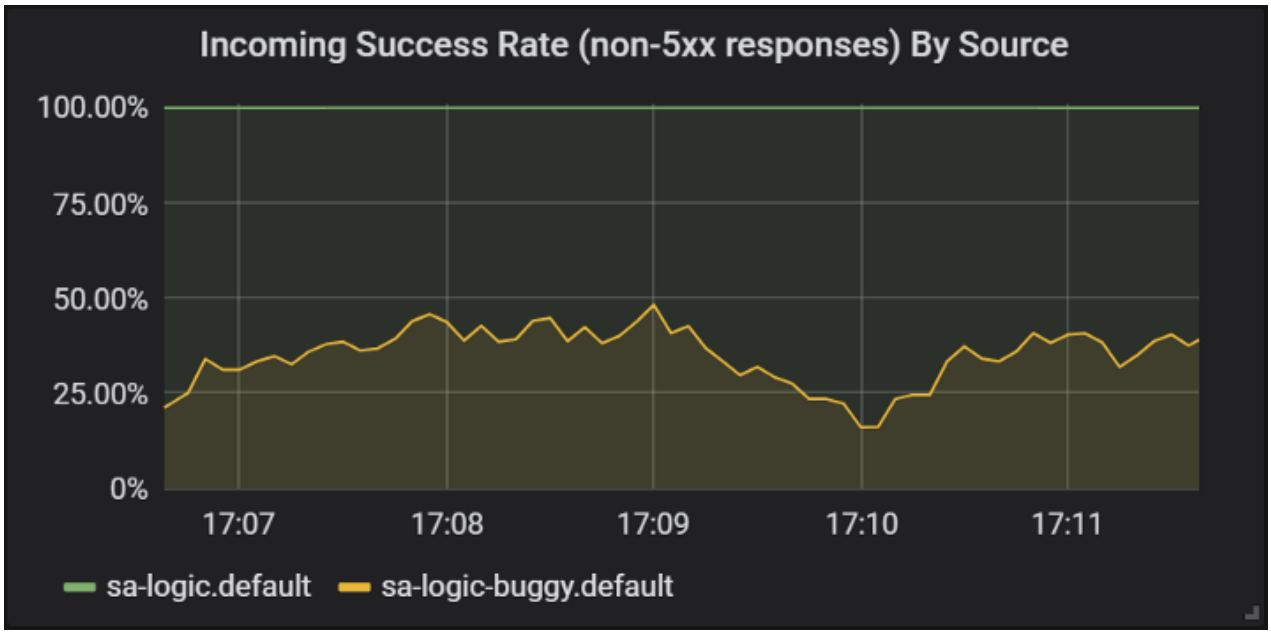

Рисунок 2.15- Успешность ответов разных версий сервиса sa-logic

Здесь впервые можно увидеть, как VirtualService применяется по отношению к Envoy'ям сервисов: когда sa-web-app делает запрос к sa-logic, он проходит через sidecar Envoy, который — через VirtualService — настроен на маршрутизацию запроса к подмножеству v1 и зеркалированию запроса к

подмножеству v2 сервиса sa-logic.

2.4.3 Канареечные выкаты

Canary Deployment — процесс выкатывания новой версии приложения для небольшого числа пользователей. Его используют, чтобы убедиться в отсутствии проблем в релизе и только после этого, уже будучи уверенным в достаточном его (релиза) качестве, распространить на б*о*льшую аудиторию.

Для демонстрации канареечных выкатов так же будет использовано подмножеством buggy у sa-logic.

Отправим 20 % трафика на версию с багами (она и будет представлять наш канареечный выкат), а оставшиеся 80 % — на нормальный сервис.

```
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: sa-logic
spec:
  hosts:
    - sa-logic
  http:
  - route:
    - destination:
        host: sa-logic
        subset: v2
      weight: 20
    - destination:
        host: sa-logic
        subset: v1
      weight: 80
```

П р и м е ч а н и е - 1 — это вес (weight), определяющий процент запросов, которые будут направлены на получателя или подмножество получателя.

Снова создав искуственный трафик получается 20% сбоя:

Time: 0.153075s Status: 200
Time: 0.137581s Status: 200
Time: 0.139345s Status: 200
Time: 30.291806s Status: 500
Time: 0.1422731s Status: 200

VirtualServices активируют канареечные выкаты: в данном случае сузились потенциальные последствия от проблем до 20% от пользовательской базы. Теперь в каждом случае, когда нет уверенности в сервисе, можно

использовать зеркалирование и канареечные выкаты.

2.4.4 Таймауты и повторные попытки

Не всегда проблемы оказываются в коде. В списке из «8 заблуждений в распределённых вычислениях» на первом месте стоит ошибочное мнение, что «сеть надёжна». В действительности сеть не надёжна, и по этой причине нам нужны таймауты *(timeouts)* и повторные попытки *(retries)*.

Ненадёжность сети попробуем симулировать случайными сбоями. Пусть сервис с проблемами имеет 1/3 вероятность на слишком долгий ответ, 1/3 — на завершение с ошибкой Internal Server Error и 1/3 — на успешную отдачу страницы.

Для того, чтобы смягчить последствия от подобных проблем и сделать жизнь пользователей лучше, можно:

-добавить таймаут, если сервис отвечает дольше 8 секунд;

-предпринимать повторную попытку, если у запроса происходит сбой.

```
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: sa-logic
spec:
  http:
  - route:
    retries:
      attempts: 3
      perTryTimeout: 3s
    timeout: 8s
    - destination:
        host: sa-logic
        subset: v1
      weight: 50
    - destination:
        host: sa-logic
        subset: v2
      weight: 50
  hosts:
    - sa-logic
```

П р и м е ч а н и я:

1 Таймаут для запроса установлен в 8 секунд

2 Повторные попытки запросов предпринимаются по 3 раза

3 И каждая попытка считается неудачной, если время ответа превышает 3 секунды.

Так мы добились оптимизации, поскольку пользователю не придётся ждать более 8 секунд и предпринимается три новые попытки получить ответ в случае сбоев, повышая шанс на успешный ответ.

Создав искусственную нагрузку проверяем в графиках Grafana, что количество успешных ответов стало свыше:

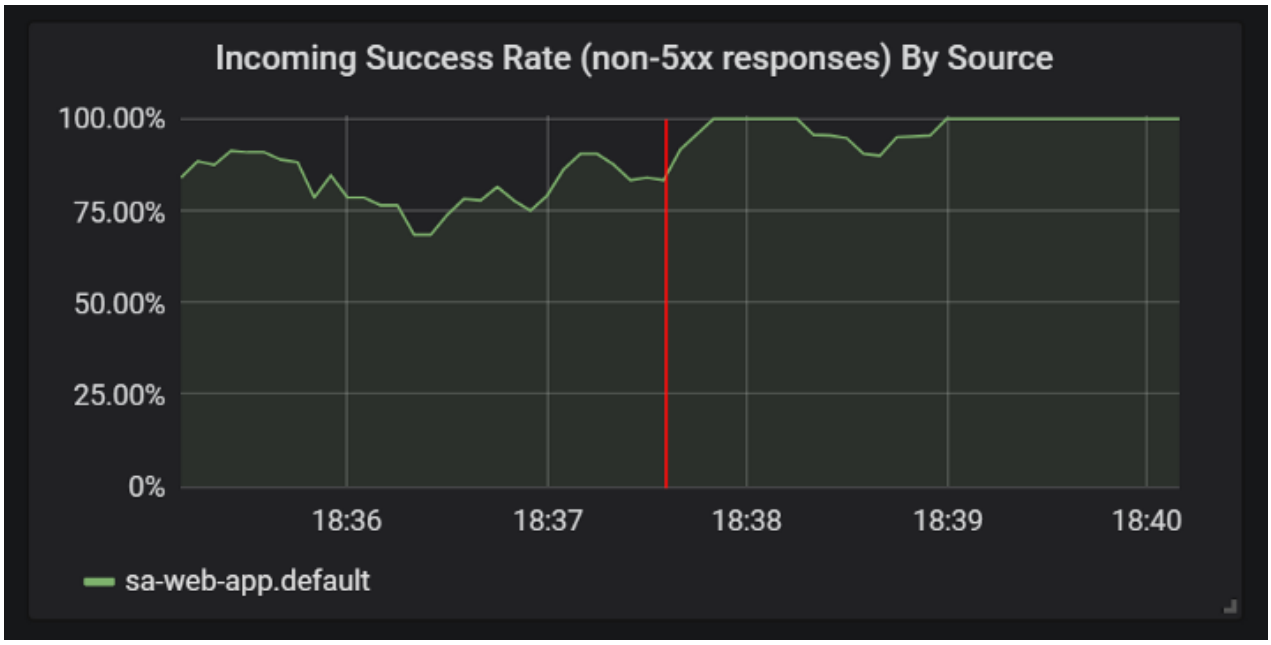

Рисунок 2.16- Улучшения в статистике успешных ответов после добавления таймаутов и повторных попыток

-*Паттерны Circuit Breaker и Bulkhead*: речь идёт о двух важных паттернах в микросервисной архитектуре, которые позволяют добиться самостоятельного восстановления *(self-healing)* сервисов;

-*Circuit Breaker («размыкатель цепи»):* используется для прекращения запросов, поступающих на экземпляр сервиса, который считается нездоровым, и его восстановления в то время, как запросы клиентов перенаправляются на здоровые экземпляры этого сервиса (что повышает процент успешных ответов);

-*Bulkhead («перегородка»):* изолирует сбои в сервисах от поражения всей системы. Например, сервис B сломан, а другой сервис (клиент сервиса B) делает запрос к сервису B, в результате чего он израсходует свой пул потоков и не сможет обслуживать другие запросы, даже если они не относятся к сервису B.

3 Преимущества Kubernetes

В наши дни важным критерием эффективности ИТ- сферы является то, насколько быстро выходят апгрейды. Это требует затрат ресурсов каждый день на то, чтобы вносить различные корректировки, такие как: введение

новых задач и модификаций, прописывание документов и скриптов на должном, качественном уровне. Помимо всего перечисленного важно, чтобы все работало в онлайн- режиме беспрерывно.

Все эти достоинства являются неотъемлемой частью Kubernetes- он использует в своем арсенале инструментов небольшие сервисы и контейнеры, с помощью которых организует работу всей структуры.

Если рассмотреть основную цель апгрейдов в ИТ- сфере, то можно отметить такой момент: важно, чтобы ИТ- разработка работала без ошибочно уже как конечный продукт. Существует ряд преимуществ работы с контейнерами, которые позволяют устранить разницу при отладке программной части и материальных серверов, избегая при этом возникновения перебоев в системе. За это отвечают те самые контейнеры, которые консолидируют все части программы в единое целое и отделяют его от прочей среды. Таким образом, можно легко открывать приложения на различных платформах для внесения изменений в их коды. Kubernetes организует работу контейнеров, с помощью которых упрощается работа при внесении изменений, а также наблюдение за корректностью работы приложений, и при необходимости, исправлении ошибок. Также, стоит отметить, что при внесении изменений, пользователи приложений не испытывают никаких неудобств.

Есть ряд причин, которые показывают, что пришло время обратиться к возможностям Kubernetes:

- в случае, когда инструмент бизнеса является основной составляющей проекта, и не может терпеть остановки в работе несмотря ни на что;
- при больших нагрузках на систему, которая в любом случае должна реагировать стремительно на все изменения и корректировки;
- когда существует дополнительная потребность в быстром увеличении ресурсов системы;
- когда существующая структура требует больших временных затрат на внедрение требуемых изменений, вместо мгновенного реагирования.

К вышеперечисленным достоинствам Kubernetes можно добавить:

- система позволяет решать задачи системно, по единому стандарту, при этом автоматизируя большую часть работы. То есть это новый подход, в котором отсутствует «ручная» работа: не нужно прописывать скрипты на каждый отдельный случай;
- отпадает зависимость от отдельных людей, которые владеют информацией о системе единолично;
- в рамках одной компании, все ИТ-системы взаимосвязаны.

Компании, которые нуждаются во всех перечисленных изменениях, могут при обращении к Kubernetes содержать свои системы в онлайн режиме бесперебойно.

В ИТ- сфере существует большое количество аналогичных платформ как Kubernetes, но именно он является основополагающим стандартом, который доступен многим. Ниже приведены заключающие подтверждения

того, что Kubernetes отвечает всем современным запросам, предъявляемым в настоящее время в ИТ- отрасли:

- нацеленность на работу с приложениями;
- удобство в использовании при работе с одним и более дата-центрами;
- существование доступной обучающей и технической документации о Kubernetes;
- работа с контейнерами Docker.

В плане преимуществ использования в работе Kubernetes с точки зрения ведения бизнеса:

- быстрота и гибкость работы системы; автоматизация работы даже при введении изменений;
- выборочная работа с отдельными частями системы для начала использования.

Для большего понимания, как построена работа от начала и до конца, важно помнить, что платформа Kubernetes избавляет от рутинной работы, беря на себя цикличные задачи. При этом отпадает необходимость в большом количестве исполнителей, поскольку все этапы проходят автоматизировано. При этом обновления могут проводиться в любое время, а происходящие изменения никак не будут влиять на текущую работу системы. Все это может находиться на физических серверах либо с использованием облачных технологий от таких поставщиков как Amazon или Azure. Существует еще огромное количество возможностей по оптимизации работы с помощью Kubernetes, за счет его технических расширений или разработки языке программировании.

3.1 CI/CD в Kubernetes

Под CI/CDL/CDP (Continuous Integration, Continuous Delivery, Continuous Deployment) обозначается автоматизация процесса над исходным кодом с доведением его до эксплуатационной среды. Каждому делению по средам выкатки соответствует свой пайплайн запуска. CI всегда только на разработческой среде. CDP всегда запускается только на эксплуатационной среде.

Огромное количество компаний сопровождают свою инфраструктуру средствами известными более 10 лет, но с развитием IT такой подход становится не эффективным и затратным. За последние 5 лет в эксплуатации инфраструктуры появляется все больше способов и инструментов для решения задач быстрее и эффективнее. К таким инструментам относятся:

- системы управления конфигурациями – ansible, puppet, chef, saltstack;
- контейнеры – docker, rkt, LXC, mesos containers, containerd;
- системы оркестрации контейнеров – kubernetes, swarm, mesos, nomad;
- системы мониторинга и логирования – prometheus, zabbix, ELK, grafana.

В зависимости от задачи применяются свои подходы использования инструментов описанных выше. Общую группу DevOps можно разбить на несколько основных:

а) инфраструктурный DevOps;

б) сетевой DevOps;

в) платформенный DevOps.

Если при первых двух более актуален подход описания инфраструктуры в виде кода(IaC) то в последнем навыки развертывания приложений по принципу CI/CD вынося это в Pipeline (gitlab, Jenkins, TravisCI и т.д.) и в отличии от IaC здесь заранее готовят образы, пакеты, обновления в зависимости от версии а хранение происходит на серверах артифактори и при развертывании не тратиться время на генерацию готового продукта.

Текущий подход эксплуатации инфраструктуры все активнее меняется в сторону контейнеров и самым популярным лидером среди систем оркестрации является Kubernetes, который нацелен на ускорение и упрощение развертывания приложений. А при использовании дополнительных наборов инструментов таких как Helm, gitlab, CI/CD, Prometheus и т.д. процесс становиться автоматизированным и требует меньшего количества обслуживающего персонала. Достаточно крупные мировые компании как CERN, Ebay, Huawei и т.д. уже используют Kubernetes в продуктивной эксплуатации.

Суть идеи инновации в применении новых подходов и инструментов в компаниях, но процедура требует хорошей подготовки технического персонала и времени на трансформацию инфраструктуры, так как нет одного идеального пути и по сути каждое внедрение это уникальный процесс для каждой компании. Преимущества от применения новых подходов:

- сокращение времени развертывания продуктов компании;

- IaC не требует написания документации, так как код- уже документация и всегда в актуальном состоянии;

- меньше затраты на ресурсы (материальные и человеческие).

Для написания автоматизированного конвеера необходимо проанализировать входные данные:

1. Какое количество сред необходимо внести в конвейер, и какие особенности присущи для каждой из них. Какие общие характеристики эти среды имеют в отношении одного сервиса.

2. Каким образом проверяеться исходники и запущенные сервисы. Используеються ли определенные приложения для этого процесса и возможен ли его перенос на какой нибуть раннер(часто из за проблем с лицензией такая опция не доступна)

3. Какие разграничения доступов необходимы. От этого зависит масштаб развертывания и интеграция с системами, и что немаловажно безопасность. В случае выхода сервиса даже для малого круга лиц, при наличии уязвимости под риском оказываеться весь интеграционный стек.

4. Как выглядит дизайн сервиса. Необходима ли интеграция со сторонними сервисами.

Стандартный стек в часто встречаемых проектах:

- Jenkins, GitLab;
- RKT, Docker, docker-compose;
- Swarm, Kubernetes;
- Helm, Prometheus, Grafana.

Пример пайплайна с иллюстрацией применения стека в типичном проекте :

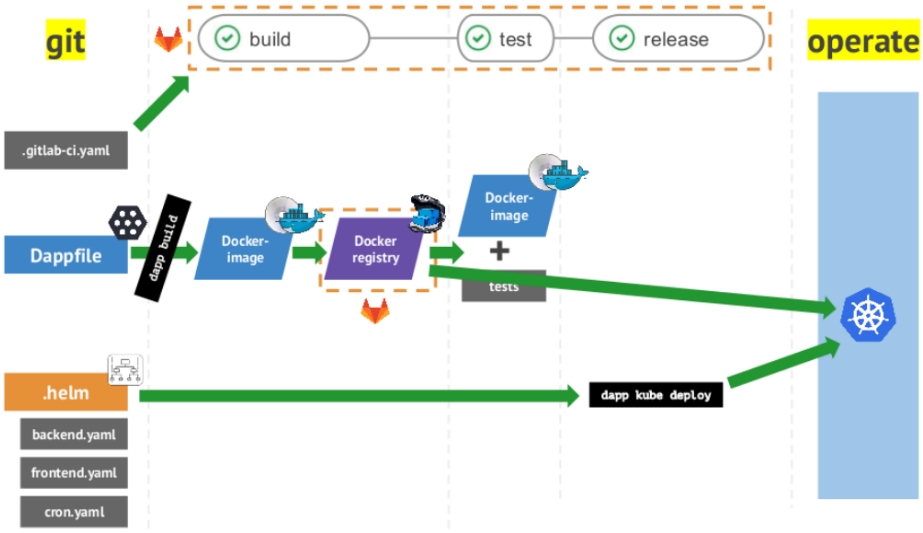

Рисунок 3.1- Deployment workflow

Микросервисы взаимодействуют по сети друг с другом передавая полезную нагрузку, и пока будет не доступен хоть один сервис говорить о полном предоставлении сервиса говорить не приходиться. И к сожалению такое происходит в самые ответственные моменты как исправление критических изьянов. А причиной тому являеться не продуманного дизайна выкатки релизов. Как пример простои из за последовательного отключения работающих сервисов и запуск новых по завершению предыдущего шага. Или выкатка хотфикса на скорую руку без проверки на разработческом стенде.

Сервис часто бывает не доступен из за интеграционных проблем. Поэтому выкатка релизов всегда оцениваеться в скопе со всеми сопутствующими компонентами интеграции вплоть до строк СУБД. И при обьемных наборах релиза лучше будет разбить его на независимые куски, чтобы вслучае обнаружения проблем оперативно откатить все назад не затягивая процес неуспешной выкатки.

При наличии в стеке СУБД и его неоходимости обновления вместе с релизной выкаткой. Применяеться пошаговые действия по формированию дампа, заливка дампа на конечную СУБД, миграция зависимостей и выкатка бэкенд сервисов для работы с готовой СУБД. Такой поэтапный план без

простоя для клиентов возможно реализовать несколькими путями. Самым надежным из них являеться поднятие независимых обновляемых компонентов с удалением старых после обновления.

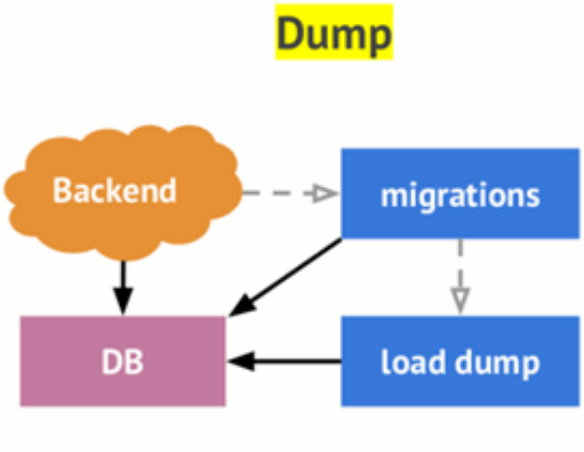

Рисунок 3.2 - Загрузка готового дампа

Во время подготовки к инициализационному развертыванию не возможно учесть всех нюансов, и крайне трудно обойтись без инсциндентов или простоев, для этого составте чеклист:

- проверить запросы каждого сервиса локально
- настройка readynes, livenes проверок для сервисов выкатываемых в Kubernetes
- запустить нагрузочное тестирование на развернутые сервисы и проверка на наличие ошибок в запросах
- правильно выставить пороги ресурсов и отслеживать алерты по сбоям во время развертывания

Рисунок 3.3- Выкат без простоя

При прогоне пайплайне CDL/CDP сбой влечет неопределенный сценарий действий который в большинстве случаев исправляеться мануально. Выяснив причины сбоя их необходимо отразить в базе знаний или править соответствующий блок пайплайна или релиза. Так как Kubernetes работает в декларативном режиме, состояние последнего развернутого манифеста приведет сервис к желаемому виду.

При проблемах иного характера инструменты как Helm позволяет вернуться на предыдущую версию выкатки релиз с минимальными действиями со стороны специалста. А так же часто применяеться логика заложенная в пайплайне по удалению ошибочных релизов без выяснения причин ошибок, что не лучший пример для использования.

Для динамических окружений(среды которые живут не продолжительное время и удаляються после достичения определенных целей) на уровне Kubernetes легко разделять среды на пространства имен, однако нужно понимать что это очень ресурсоемкое решение, так как по сути воспроизводиться новая среда с таким же обьемом ресурсов как целевая среда. Но они оправдана в случаях когда нельзя или невозможно выполнять операции на целевом окружении. Как например нагрузочное тестирование(приводит к простою сервиса из за неспособности обслужить непомерный обьем нагрузки), проверка на отказоустойчивость(приводит к простою сервиса для проверки действия системы в случае отказов).

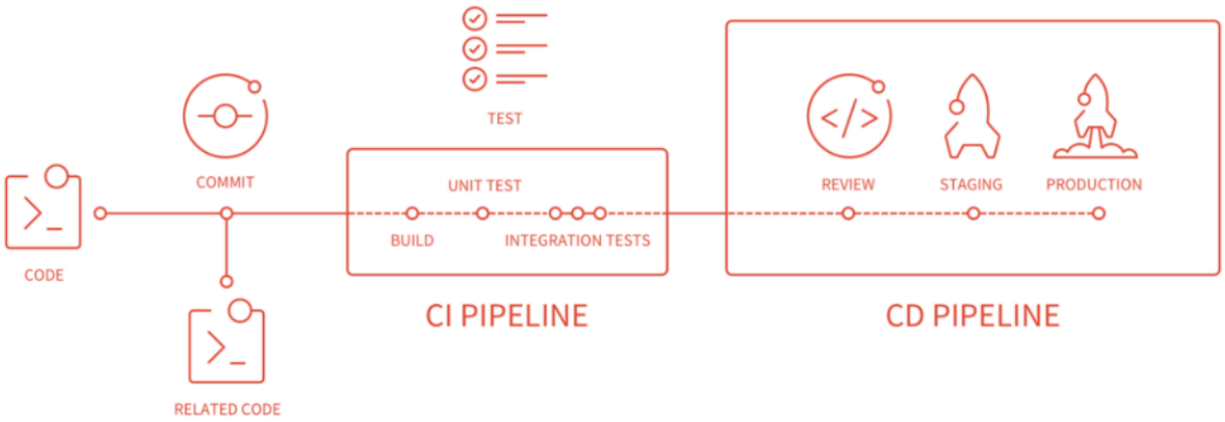

Рисунок 3.4- Пайплайн

Пайплайн – последовательность выполнения группы задач, которая разбита по логическим признакам. На примере рисунка выше представлен простой пайплайн, который реализуеться на системах как Gitlab-CI, Teamcity, Jenkins и т.д. в зависимости от предпочтений каждой группы автоматизаторов. Описание основных блоков пайплайна:

- build - сборка сервиса или образа контейнера
- test – статанализ, junit, автотесты, и т.д.

- staging - окружение для развертывания релиза с целью тестирования или разработки нового функционала

- pre-production - окружение для развертывания релиза в дубликат от промышленной с идентичными ресурсами

- approve - ручная кнопка запуска пайплайна для дальнейшего развертывания релиза в промышленную эксплуатацию

- production - окружение для развёртывания релиза в эксплуатацию.

Последовательность исполнения каждой стадии в деталях:

1) При любом собитии push в git репозиторий срабатываеть вебхук который ссылается на API сервис пайплайна раннера и триггерит запуск нужного пайплайна по входным параметрам полученным из запроса. В зависимости от настроек пайплайне будет исполняться на определенном агенте(среда исполнения пайплайна) где возможно преднастроены необходимые инструменты для исполнения задач. Выбор агента может быть по умолчанию(обязательно хоть один агент должен присутсвовать, иначе задачам негде будет исполняться) или по меткам описываемые независимо для каждой стадии.

2) На стадии build, стадия запускаеться на агенте в разработческой среде с набором инструментов для сборки сервиса из исходного кода и паковкой в docker контейнер. А так же формируеться файл со значениями для подстановки в Helm шаблоны на все окружения для развертывания. И в виде артефактов заливаються в службу регистри и файловое хранилище с определенной версионностью. А по прохождению статического анализа исходников генерируеться пороги качества(QG - Quality Gate) с требуемым минимальными значениями для оценки годности текущей сборки на предмет его дальнейшего движения дальше по другим стадиям. Пороги качества так же сохраняються на центральном хранилище для ведения истории.

3) На стадии test производиться тестирования исходников для выявления каких либо проблем, так как чем раньше она находиться тем дешевле обходиться ее устранение. После каждого теста производиться дозапись в файл порога качества хранимая на центральном репозитории. В этой стадии допускаеться развертывание сервиса на разработческом или тестовом контуре, чтобы провести тестирование непосредственно самого сервиса на работающие кластере. Как например функциональное, регрессионное тестирование.

4) На стадии staging сервис развертываеться в окружении похожем на целевую по интеграционным возможностям но гораздо сокрашённом масштабе, это делаеться для проверки другого ряда тестов не доступных на предыдушем шаге. Так же по прохождению тестов производится запись в пороги качества. Начиная с этой стадии пороги качества влияют на выбор о принятии решения по развертыванию релиза в случаеш успешного прохождения предыдуших порогов качества.

5) На стадии pre-production выполняются все те же действия как на staging, а сам контур является полной копией целевого окружения. С разницей что в него не поступает реальный трафик клиентов.

6) Стадия approve необходим как предохранитель для ручного управления процессом развертывания в целевую среду, и связано это с тем что предыдущая стадия требует некоторое время(до нескольких дней) для проверок систем.

7) Последняя стадия production выполняет выкатку релиза на целевое окружение где на сервисы поступает реальный трафик пользователей. По ее завершении пайплайн начинает цикл снова для другого релиза

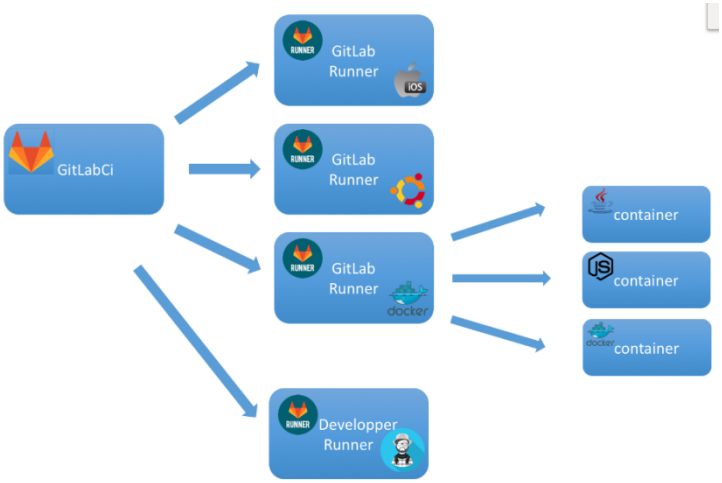

Рисунок 3.5 - Ноды gitlab runner

Для удобства работы с огромными .gitlab-ci.yml файлами целесообразно разбить этот файл на части и перенести часть кода в другие файлы с последующей ссылкой к ним. Это повышает читаемость пайплайна и переиспользовать повторяемые куски текста. А так же делать скрытыми некоторые стадии добавляя перед словом точку, но не теряя способности ссылаться из основного пайплайна

Тестовый пайплайн для прогона Junit тестов как на примере выше будет запущен на стадии kymbat-testing с меткой раннера kymbat-deploy для бранчей devops и junit:

```
junit:
  stage: CI
  tags: [kymbat-deploy]
  script:
    - print " junit test"
  only:
    - tags
    - /^junit-.*$/
```

- /^devops-.$/*

Во время выполнения команды Helm в качестве аргументов указывается имя развертываемого чарта , директория содержащий чарт шаблоны, опционально указывается файл со значениями для подставления в шаблон.

Чарты так же доступны в репозиториях Helm, с инструкцией к применению и примерами по развертыванию с измененными входными параметрами.

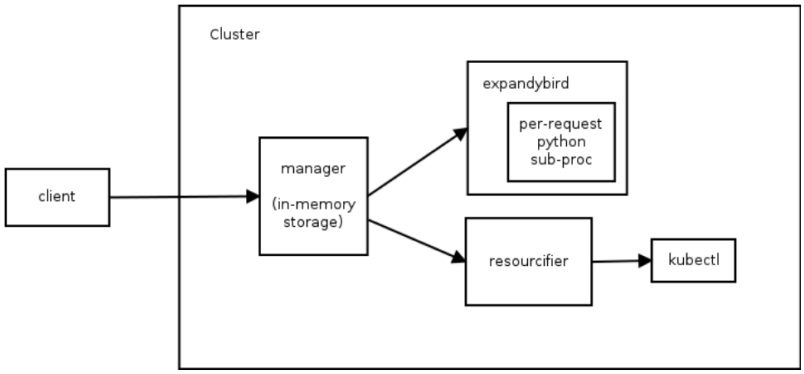

Рисунок 3.6 - 3-х компонентная а*рхитектура Helm*

- Manager(он же tiller) запущен как DeploymentSet в кластере и слушает API запросы от Helm клиента. Канал рекомендуется шифровать сертификатами генерируемыми в кластере с включенным Role Based Access. Индивидуально для каждого пользователя. И при просмотре истории выкатов легко отслеживаться сотрудник запустивший команду над определенным релизом в определенное время.
- Resourcifier вы полняет роль испольнения полученного манифеста через команду kubectl

В Helm позволяет обновлять, удалить, откатить на раннюю версию релиза специальными аттрибутами при запуске команды. Что по сравнению с модификациями в манифесте гораздо быстрее выполнить. А при обновлении под замену обьектов подпадают только те что имеют изменения. Касательно настроек самого Helm-а доступно обновлением ConfigMaps-а в кластере Kubernetes.

Главный акцент это сокращение описываемых обьектов Kubernetes в манифестах в формате yaml в десятки раз, за счет шаблонной заменты значений переменных в файлах. А так же повторное переиспользование шаболонов сильно сужает обьем текстовой информации.

apiVersion: v1
kind: Pod

```
metadata:
  name: {{ template "alpine.fullname" . }}
  labels:
    heritage: {{ .Release.Service }}
    release: {{ .Release.Name }}
    chart: {{ .Chart.Name }}-{{ .Chart.Version }}
    app: {{ template "alpine.name" . }}
spec:
  restartPolicy: {{ .Values.restartPolicy }}
  containers:
  - name: waiter
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    command: ["/bin/sleep", "9000"]
```

Helm чарты хранят в себе один и более сервисов в виде кода и занимают небольшое место на хранилище, то есть по сути являются метаданными для развертывания сервисов.

```
helmchart
|-- Chart.yaml
|-- templates
|   |-- NOTES.txt
|   |-- _helpers.tpl
|   |-- deployment.yaml
|   |-- ingress.yaml
|   `-- service.yaml
`-- values.yaml
```

Если посмотреть на структуру директории, то в нем содержатся файлы Chart.yaml, values.yaml и директория templates, где в файле Chart.yaml с минимальным количеством опций:

```
apiVersion: версия API чарта
name: Имя чарта
version: версия чарта
```

Для лаконичности здесь были указаны не все опции доступные для настроек чарта, но перечисляя самые популярные, такие как: источник хранения чарта, зависимости для развертывания чарта, тэги, аннотации, информация об авторе.

В файле values.yaml прописываются значения для переменных определенные в папке файлов templates. Существуют методы для описания

данного файла для повышения читаемости кода и способности его редактирования.

В директории templates хранятся файлы YAML формата, описывающие манифесты Kubernetes, с переменными для подстановки из файла values.yaml. С помощью языка шаблонов можно составлять чарты сложной конфигурации и версионировать его как готовый релиз. Хранение списка релизов централизировано упрощает работу в эксплуатации. Также чарт репозитории могут быть скопированы из других источников. Таким образом, нет необходимости изобретать свое решение, а достаточно скопировать готовый чарт и развернуть сервис со своими параметрами, описанными в файле Chart.yaml. Или на основе готового чарта написать сервис для собственных нужд с минимальными временными затратами, а версионирование позволяет улучшить релиз с течением времени.

При описании values.yaml необходимо описывать образы контейнеров, а так же учетные данные для доступа к ним, самый простой и надежный способ указания секретный данных – прописать их в Kubernetes Secret-ы. А в файле значений указывать название секретного хранилища, с помошью которого из узлов кластера будут скачиваться образы контейнеров. Образы хранятся на схожем с чарт репозитариями – серверах регистри или репозиториях образа контейнеров. Gitlab имеет собственный регистри сервис, что очень удобно при совместной работе с пайплайнами, так как трафик передачи будет локальным, и время выполнения пайплайна меньше, чем если бы использовался внешний регистри-сервис. К тому же пайплайн будет использовать общие переменные для регистри и хранилища коды.

На рисунке 32 изображены компоненты участвующие в развертывании сервиса:

- Helm client
- Helm server(Tiller)
- Репозиторий чартов
- Кластер Kubernetes
- Helm чарт
- Kubernetes манифесты

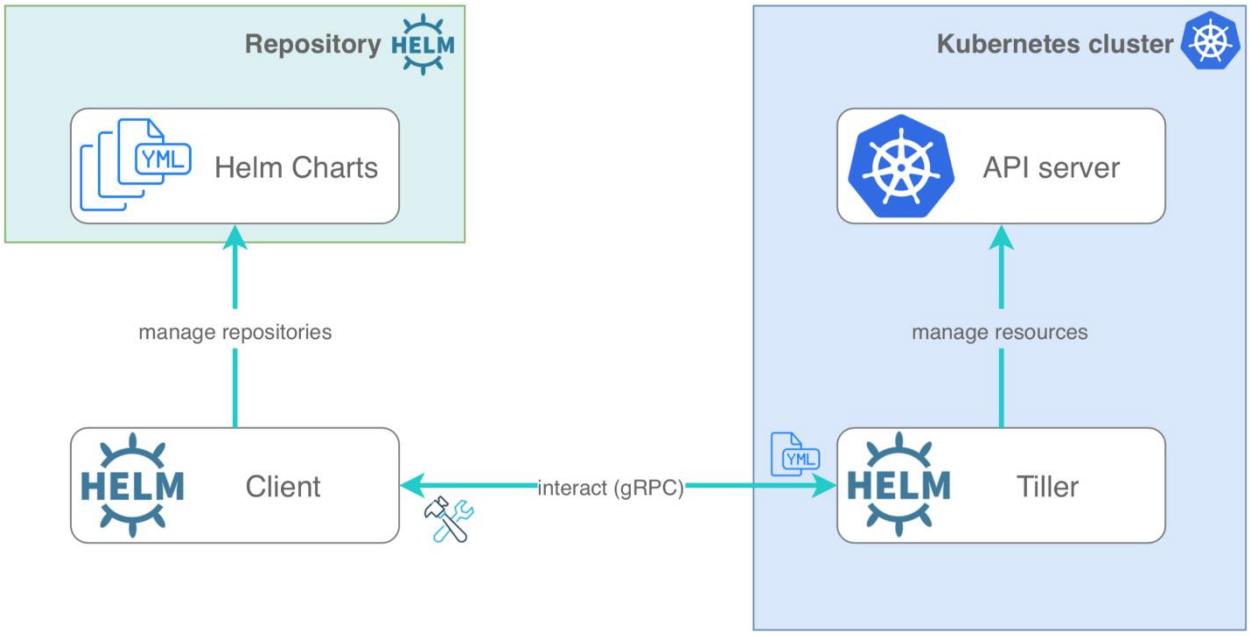

Рисунок 3.7 - последовательность развертывания с помощью *Helm*

Клиент helm-а устанавливается на gitlab-runner или другом узле откуда необходимо устанавливать или отслеживать релизы на кластере. С помощью клиентской программы выполняются операции над чартами – загрузка, создание, установка, удаление, проверки и т.д. Но главная функция- это преобразование чарта в манифесты Kubernetes, которые отправляются на Helm сервер (Tiller). Передача манифестов по умолчанию не шифруется, но есть возможность обезопасить передачу с помощью сертификатов, в настроечном файле helm клиента инициируется запрос на добавления сертификата и Tiller добавляет его в кластер, и все последующие данные передаются по шифрованному каналу. Tiller получив манифест пытается установить его в течении таймаута пяти минуты по умолчанию. И сохраняет историю событий по релизам в кластере, доступные для просмотра из клиента. В случае необходимости можно откатить релиз или переустановить с новыми параметрами всего одной командой.

Образы контейнеров попадают в сервис регистри уже в готовом для употребления виде, в концепции Immutable architecture, изменяемые данные внутри контейнера не сохраняются локально после перезагрузки, на самом деле контейнер после перезагрузки обнуляется. Каким же образом контейнеры получаются в том виде, в котором его используют? Они собираются с помощью программы docker с опцией build из файла Dockerfile по умолчанию.

From ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py

Это пример простого контейнера на базе ОС ubuntu 18.04 версии. На второй строке контейнер копирует все содержимое локальной директории в папку /app в будущий образ. Третьей строкой выполняется команда внутри будущего образа для сборки директории /app, а в самом конце указывается запуск файла /app/app.py интерпретатором python. Собранный образ отправляется в регистри командой docker с опцией push и заранее проставленным тэгом. По распространенным методикам создания контейнера рекомендуется создавать образы с минимальным набором излишеств, только необходимые для работы компоненты должны быть включены в образ. Самым простым способом сократить размер образа – использовать ОС минимального размера. Проверить собранный образ можно без его запуска на кластере, на локальном компьютере. Это позволяет протестировать сервис на более раннем уровне, а рано найденная ошибка уменьшает потраченное время на исправление на позднем этапе.

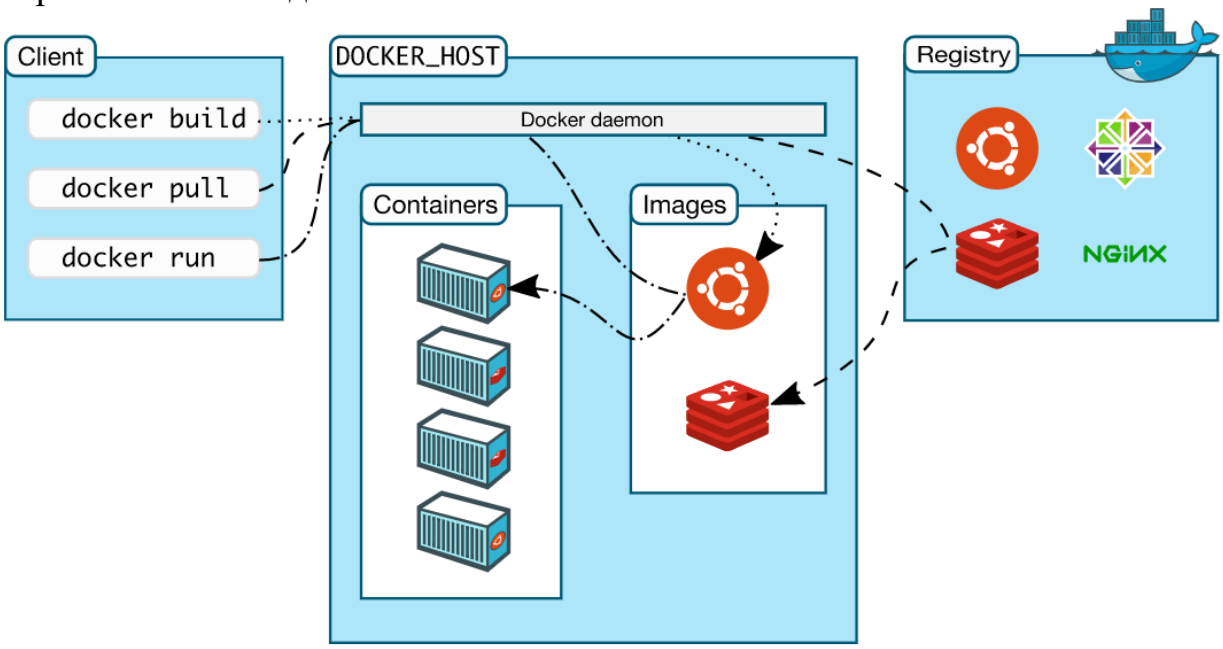

Рисунок 3.8 - Процессы работы с docker

На рисунке выше приведен пример создания и запуска образа контейнера. Слева команды создания, скачивания и запуска образа. Колонка посередине описывает процесс запуска и выгрузки с картины справа.

Заключение

Kubernetes и контейнерная технология – современные инструменты для проведения исследований в области науки о жизни. Большинство рабочих процессов на проектах с открытым исходным кодом, которые могут быть использованы для создания пользовательских образов контейнеров и приложений для использования с Kubernetes, который показал свою эффективность, надежность и масштабирование. В средах HPC, таких как UPPMAX, количество установленных программ в области науки и жизни с различными версиями будет увеличиваться. Используя контейнеры, программное обеспечение не нужно устанавливать в среде, а запускать контейнерами и переиспользовать для других пользователей. Kubernetes может работать на разных платформах: от вашего ноутбука до виртуальных машин на облачном провайдере до стойки из металлических серверов. Усилия, необходимые для настройки кластера, варьируются от запуска одной команды для создания собственного настраиваемого кластера.

Я рассмотрел основные абстракции Kubernetes, а также другие сопутствующие сервисы дополняющие его работу, такие как мониторинг, пакетный менеджер, журналирование и безопасность. Детально было изучено несколько вариантов управления трафиком с помошью Istio, и исследованы манифесты запуска и графики синтетической нагрузки. Дано описание преимущества использования Kubernetes как с технической, так и с экономической точки зрения. Были затронуты требования к инженеру помимо технических навыков – “soft skills” необходимые для продуктивного взаимодействия команды при работе с Kubernetes, а также DevOps практики используемые отрасли и сопутствующие методологии и автоматизации, как CI/CD дополняющие концепцию как единое решение.

На сегодняшний день в Казахстане Kubernetes обретает все большее количество сторонников и уже около 30 компаний на момент написания данной работы используют его в промышленной эксплуатации. Причиной тому являются преимущества в скорости и экономической доступности решения. Но с другой стороны к инженерам предъявляются более высокие требования что, на мой взгляд, является положительным явлением.

Существует сертификация для специалистов по Kubernetes – Certified Kubernetes Administrator(CKA), Certified Kubernetes Application Developer(CKAD), наличие сертификата говорит о наличии у его владельца знаний по работе с Kubernetes. Сертификация была создана фондом облачных вычислений(CNCF).

Kubernetes так же представлен и облачных архитектурах как AWS, GCP, Azure. И с каждым днем ценники на эти ресурсы становятся доступнее, а функционал расширяется. Удобство такого сервиса заключается в гарантировании облачным провайдером доступности его узлов. Широкий набор интегрируемых облачных сервисов может заменить наличие собственного оборудования в датацетрах, и в любой момент, когда ресурсы

простаивают, можно составить правило для отключения этих узлов, уменьшая расходы компании. Официально Amazon EKS анонсировала SLA в 99.9% в промежутке времени в один месяц. Чего довольно трудно добиться в датацентах Казахстана по разным техническим причинам.

Список литературы

1 Marko Luksa, “Kubernetes in Action” – Manning, 2017 - 624c.

2 Kelsey Hightower, Brendan Burns, Joe Beda, “Kubernetes: Up and Running: Dive into the Future of Infrastructure 1st Edition” – O’Reilly Media, 2017 - 272c.

3 Nigel Poulton, “Docker Deep Dive: Zero to Docker in a single book” – Leanpub, 2017 - 251c.

4 Lee Calcote, Zack Butcher, “Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe 1st Edition” - O’Reilly Media, 2019 - 272c.

5 Christian E. Posta, Rinor Maloku, “Istio in Action” – Manning, 2018 – 375c.

6 Bilgin Ibryam, Roland Hub, “Kubernetes patterns” - O’Reilly Media, 2019 - 266c.

7 JamesTurnbull, “The Terraform book” – 2016 - 318c.

8 David Farley, Jez Humble, “Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation” – 2010 - 734c.

9 https://prometheus.io/docs/introduction/overview

10 https: //kubernetes.io/docs/home/

11 https://docs.fluentd.org

12 https://habr.com/ru/company/flant/blog/438426

13 https://habr.com/ru/company/flant/blog/331188

14 https://habr.com/ru/company/flant/blog/346304/

15 https://habr.com/ru/company/flant/blog/440378/

16 https://habr.com/ru/company/flant/blog/443668/

Аббревиатуры

ОС – Операционная система
ЦП – Центральный процессор
PaaS – Platform as a Service
IaaS – Infrastructure as a Service
API – Application programming interface
VM – Virtual machine
REST – Representational state transfer
SCTP – Stream control transmission protocol
Json – JavaScript object notation

```
stages:
  - build
  - CI
  - CDL
  - CDP
  - approve
  - Prom
## build stage
build:
  stage: build
  tags: [deploy]
  script:
    - echo "Build"
## CI stage
junit:
  stage: CI
  tags: [kymbat-deploy]
  script:
    - echo " junit"
test integration:
  stage: CI
  tags: [deploy]
  script:
    - echo "test integration"
test selenium:
  stage: CI
  tags: [deploy]
  script:
    - echo "test selenium"
## CDL stage
.CDL-deploy: &CDL-deploy
  tags: [deploy]
  stage: CDL
  when: manual
  script:
    - echo $CI_BUILD_NAME
deploy to dev-1:
  <<: *CDL-deploy
deploy to dev-2:
  <<: *CDL-deploy
deploy to devops-1:
```

```
  <<: *CDL-deploy
deploy to devops-2:
  <<: *CDL-deploy
deploy to qa-1:
  <<: *CDL-deploy
deploy to qa-2:
  <<: *CDL-deploy
## CDP stage
deploy to CDP:
  stage: CDP
  tags: [deploy]
  when: manual
  script:
    - echo "deploy to CDP"
## approve stage
approve:
  stage: approve
  tags: [deploy]
  when: manual
  script:
    - echo "APPROVED"
NOT approve:
  stage: approve
  tags: [deploy]
  when: manual
  script:
    - echo "NOT APPROVED"
## Prom stage
deploy to Prom:
  stage: Prom
  tags: [deploy]
  when: manual
  script:
    - echo "deploy to Prom!"
```

Обязательные поля в каждой задаче:
- stage: стадия в которую собраны набор выполняемых тасков
- script: выполнение набора команд на локальном интерпритаторе CLI
- when: условие для запуска стадии
- tags: метка для определения Gitlab-Runner на котором будет запущена стадия