МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра компьютерных технологий и систем

АНАЛИЗ И РАЗРАБОТКА РАСПРЕДЕЛЁННОЙ АРХИТЕКТУРЫ ЭКСПЛУАТАЦИИ НЕЙРОННЫХ СЕТЕЙ

Отчет по преддипломной практике

Ларина Егора Сергеевича
студента 4-го курса
«Информатика»

Научный руководитель:
старший преподаватель С. В. Шолтанюк

Минск, 2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последние годы микросервисная архитектура значительно приобретает популярность в области разработки программного обеспечения. Основная идея микросервисов заключается в разбиении сложных программных систем на отдельные, слабо связанные компоненты с целью обеспечения гибкости, упрощения процессов разработки и масштабирования. Однако, эффективное управление такими многочисленными и взаимодействующими сервисами предполагает наличие централизованного описания и систематизированного подхода к их разработке и поддержке.

Целью данной работы является разработка масштабируемой и надежной архитектуры для инференса изображений с помощью нейронных сетей.

В процессе исследования рассмотрены существующие методы, инструменты и подходы, применяемые для проектирования микросервисных архитектур и разработана система, удовлетворяющая критериям микросервисной архитектуры для генерации изображений из пользовательских запросов.

ГЛАВА 1

Генерация изображений с помощью методов машинного обучения

1.1 U-Net

1.2 Диффузионные нейронные сети для генерации изображений

1.3 Латентные диффузионные нейронные сети для генерации изображений

Генерация изображений с помощью нейронных сетей сегодня играет важную роль в развитии информационных технологий, позволяя автоматизировать создание иллюстраций. Широкое распространение генеративных технологий стало возможно благодаря улучшению методов обучения нейронных сетей, которые решают данную задачу, и развитию вычислительной техники.

Одним из значимых методов обучения нейронных сетей, предназначенных для генерации изображений, стал Generative adversarial network (GAN). Ключевой особенностью данного метода на этапе обучения является наличие двух глубоких нейронных сетей: $G(z)$, которая по распределению p_g ставит в соответствие векторам z изображения x, и сети $D(x)$, которая возвращает скалярное значение, которое является вероятностью того, что x является реальным изображением, то есть является частью обучающей выборки. Основная идея процесса обучения заключается в минимизации величины $\log(1 - D(G(z)))$, что означает, что G пытается «обмануть» дискриминатор D, что не требует привлечения учителя.

Таким образом процесс получения распределения p_g можно представить в виде игры с нулевой суммой двух игроков с критерием

$$\min_G \max_D V(D, G) = \mathbf{E}_{x\ p_{data}(x)}[\log D(x)] + \mathbf{E}_{z\ p_z(z)}[\log(1 - D(G(z)))]$$

Данный подход гарантирует сходимость при увеличении обучающей выборки и обеспечивает хорошую скорость инференса относительно подходов, основанных на других вероятностных методах, но является очень трудоза-

тратным при обучении модели.

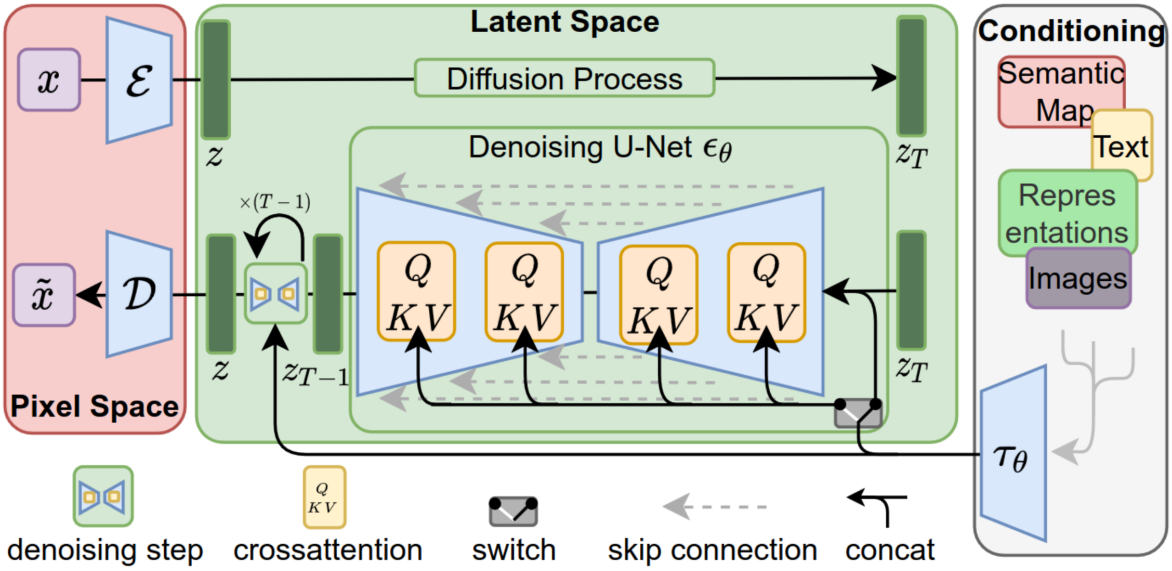

Рисунок 1.1 Схема архитектуры [1]

ГЛАВА 2

Обзор основных подходов для реализации микросервисов

2.1 Обзор микросервисной архитектуры

Представление о микросервисах родилось в 2005 году из методологии SOA путем обобщения философии UNIX для веб-компонент. Веб-сервисы сопоставляются независимым программам, комбинация которых осуществляется с помощью сетевого транспорта данных по аналогии с именованными и неименованными UNIX каналами. В отличие от SOA микросервисы не фокусируются на конкретных форматах передачи данных, как SOAP, а дают представление о том, как должны коммуницировать сервисы, чтобы при сохранении слабой связности компонент поддерживать устоявшийся контракт обмена информацией [2].

В современной литературе не существует одного конкретного определения микросервиса. Обычно микросервисы определяют через различные критерии соответствия распределенной системе с низкой связностью ее отдельных компонент.

Большое внимание в теории микросервисов уделяется вопросам масштабируемости. Для всестороннего описания масштабируемости в контексте разработки распределенных систем вводится понятие куба масшабируемости [3] (Рисунок 2.1).

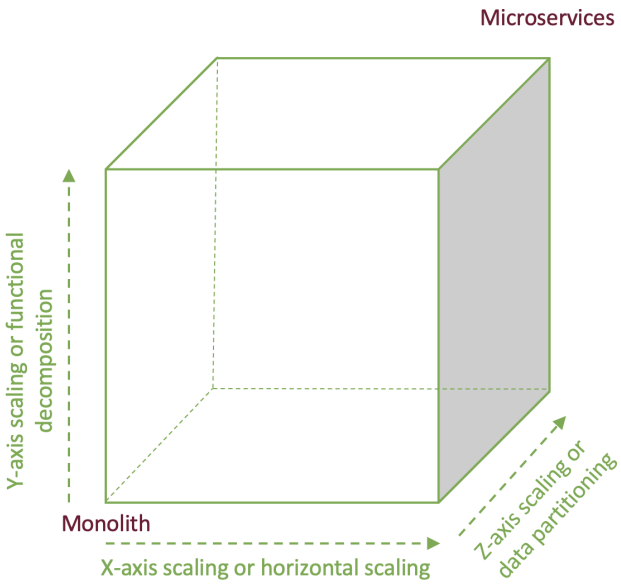

Рисунок 2.1 Куб масштабируемости

Эта модель определяет три направления для масштабирования приложений: X, Y и Z.

Масштабирование по оси X часто применяют в монолитных приложениях. Запускаются несколько экземпляров программы, размещенных за балансировщиком нагрузки. Балансировщик распределяет запросы между N одинаковыми экземплярами. Это распространенный способ улучшить производительность и стабильность приложения.

Масштабирование по оси Z тоже предусматривает запуск нескольких экземпляров монолитного приложения, но в этом случае, в отличие от масштабирования по оси X, каждый экземпляр отвечает за определенное подмножество данных. Маршрутизатор, выставленный впереди, задействует атрибут запроса, чтобы на править его к подходящему экземпляру.

Масштабирование по осям X и Z увеличивает производительность и стабильность приложения. Но ни один из этих подходов не решает проблем с усложнением кода и процесса раз работки. Чтобы справиться с ними, следует применить масштабирование по оси , или функциональную декомпозицию. То, как это работает, показано на рисунке 2.2.

Рисунок 2.2 Пример декомпозиции приложения через отдельные сервисы

Таким образом, сервис — это приложение, реализующее узкоспециализированные функции, такие как управление заказами, управление клиентами и т. д. Сервисы масштабируются по оси X, некоторые из них могут использовать также ось Z. Например, сервис Order (Рисунок 2.2 имеет несколько копий, нагрузка на которые балансируется. Одно из обобщенных определение микросервисной архитектуры (или микросервисов) звучит так: это стиль проектирования, который разбивает приложение на отдельные сервисы с разными функциями. Заметим, что размер здесь вообще не упоминается. Главное, чтобы каждый сервис имел четкий перечень связанных между собой обязанностей.

Микросервисная архитектура имеет следующие преимущества:

- Она делает возможными непрерывные доставку и развертывание крупных, сложных приложений.
- Сервисы получаются небольшими и простыми в обслуживании.
- Сервисы развертываются независимо друг от друга.
- Сервисы масштабируются независимо друг от друга.
- Микросервисная архитектура обеспечивает автономность команд разработчиков.
- Она позволяет экспериментировать и внедрять новые технологии.
- В ней лучше изолированы неполадки других сервисов.

Одна из проблем, возникающих при использовании микросервисной архитектуры, связана с отсутствием конкретного, хорошо описанного алгоритма разбиения системы на микросервисы. Что означает, что эта часть данной методологии, не имеет формального описания [2].

В связи с этим также выделяют понятие распределенного монолита.

Распределенным монолитом называют набор связанных между собой сервисов, которые необходимо развертывать вместе. Распределенному монолиту присущи недостатки как монолитной, так и микросервисной архитектуры.

Еще один недостаток микросервисной архитектуры состоит в том, что при создании распределенных систем возникают дополнительные сложности для разработчиков. Сервисы должны использовать механизм межпроцессного взаимодействия. Это сложнее, чем вызывать обычные методы в общей памяти приложения. К тому же код должен уметь справляться с частичными сбоями и быть готовым к недоступности или высокой латентности удаленного сервиса. Реализация сценариев, охватывающих несколько сервисов, требует применения дополнительных технологий.

Для реализации надлежащего уровня изоляции между микросервисами применяется паттерн database per service. В этом случае каждый микросервис имеет собственную базу данных при необходимости холодного хранилища данных. Это позволяет вносить правки в функциональность сервиса без риска сломать функциональность другого сервиса и привлечения других команд для согласования изменений, а также позволяет просто оценивать время ответов и нагрузку для каждого из сервисов по отдельности, что сложно осуществимо при использовании общих ресурсов.

Несмотря на вышеперечисленные преимущества упомянутого паттерна, database per seervice затрудняет реализацию комбинированных транзакций и запросов. Что также предполагает комбинацию данных на уровне кода микросервиса при поддержке этого в API и является еще одним недостатком микросервисных архитектур.

Также IDE и другие инструменты разработки рассчитаны на создание монолитных приложений и не обеспечивают явной поддержки распределенных приложений. Написание автоматических тестов, затрагивающих несколько сервисов, — непростая задача. Все эти проблемы характерны для микросервисной архитектуры. Вдобавок микросервисная архитектура существенно усложняет администрирование. В промышленной среде приходится иметь дело с множеством экземпляров разнородных сервисов, для успешного раз-

вертывания которых требуется высокая степень автоматизации.

Еще одна проблема микросервисной архитектуры связана с тем, что развертывание функций, охватывающих несколько сервисов, требует тщательной координации действий разных команд разработки и создание плана выкатки. Для сравнения: в монолитной архитектуре возможно параллельное обновление для нескольких компонентов.

Еще одна трудность связана с решением о том, на каком этапе жизненного цикла приложения следует переходить на микросервисную архитектуру. Часто во время разработки первой версии система не сталкивается с проблемами, которые эта архитектура решает [4]. Более того, применение сложного, распределенного метода проектирования замедлит разработку. Для стартапов, которым обычно важнее всего как можно быстрее развивать свою бизнес-модель и сопутствующее приложение, это может вылиться в непростую дилемму. Использование микросервисной архитектуры делает выпуск начальных версий довольно трудным. Стартапам почти всегда лучше начинать с монолитного приложения.

Однако со временем возникает другая проблема: как справиться с возрастающей сложностью. Это подходящий момент для того, чтобы разбить приложение на микросервисы с разными функциями. Рефакторинг может оказаться непростым из-за запутанных зависимостей. В связи с этим к переходу на них следует отнестись очень серьезно. Но обычно для сложных проектов, таких как пользовательские веб- или SaaS-приложения, это правильный выбор. Такие общеизвестные сайты, как eBay, Amazon.com, Groupon и Gilt, в свое время перешли на микросервисы с монолитной архитектуры.

При использовании микросервисов приходится иметь дело с множеством проблем, связанных с проектированием и архитектурой. К тому же многие из этих проблем имеют несколько решений со своими плюсами и минусами. Единого идеального решения не существует [2].

2.2 gRPC в разработке микросервисов

Как уже было упомянуто в предыдущем пункте, обмен данными между микросервисами имеет большое значение при проектировании распределенных систем. При правильном проектировании микросервисы сохранят свою автономность, в то же время необходимо вносить в них изменения и выпускать их новые версии независимо от остальных частей системы. При некорректном исполнении система будет функционировать со сбоями и низкой производительностью, что недопустимо при построении микросервисной архитектуры.

Для определения способа взаимодействия одного микросервиса с другим имеется выбор между текстовыми форматами данных, как JSON, XML, SOAP и т.д, и бинарными форматами данных такими, как Protocol Buffers, Flatbuffers, Cap'n Proto и Simple Binary Encoding.

При выборе формата передачи данных обычно руководствуются скоростью сериализации и десериализации, простотой использования и поддержкой со стороны сообщества и разработчиков.

Текстовые форматы данных обычно лучше всего поддержаны инструментами разработки и не требуют дополнительных усилий по внедрению их в API сервисов, однако они значительно проигрывают в скорости передачи данных и не предоставляют возможности версионирования без runtime проверок схемы данных, что несет еще большие накладные расходы при конвертации данных, несмотря на то, что за время своего существования парсеры для данных форматов, в частности JSON и XML, были сильно оптимизированы и поддержаны виртуальными машинами многих языков. Поэтому при выборе формата данных для коммуникации микросервисов обычно выбирают бинарные форматы с возможностью версионирования и строгой схемой [5].

Вышеупомянутые Cap'n Proto, SBE и Flatbuffers демонстрируют хорошую производительность в синтетических тестах, однако не обладают поддержкой широкого списка языков программирования и инструментов для разработки [6]. Protocol Buffers же предлагает разумный компромисс между производительностью и простотой использования, поэтому в настоящий момент является самым популярным вариантом при разработке микросервисов [7].

Для написания Protobuf файлов используют язык описания интерфейсов (IDL). Например, чтобы описать структуру данных сообщения, нужно добавить message, имя структуры, а внутри тип, название и номер поля. Номера

полей нужны для обратной совместимости, поэтому не стоит менять их последовательность при добавлении или удалении полей. Образец сообщения представлен в листинге 2.1.

```
syntax = "proto3";
package user;

service User {
  rpc GetUser(GetUserRequest) returns (UserInfo) {}
  rpc CreateUser(UserInfo) returns (UserStatus) {}
}

message UserInfo {
  string email = 2;
  string name = 3;
  repeated string phone = 4;

  message Address {
    string street = 1;
    string city = 2;
    string state = 3;
    string zip = 4;
  }

  Address address = 5;
}

message GetUserRequest {
  string email = 1;
}

message UserStatus {
  string error = 1;
}
```

Листинг 2.1 Сетевые хендлеры, осуществляющие основую логику

В Protocol Buffers данные представлены и передаются в бинарном формате, что существенно уменьшает время сериализации/десериализации данных. Также с применением механизма Protocol Buffers размер передаваемых данных в разы меньше [5].

В настоящее время наиболее популярным RPC-решением для интеграции микросервисов является gRPC, который по умолчанию использует Protocol

Buffers для кодирования сообщений. Это высокопроизводительный фреймворк, разработанный компанией Google для вызовов удаленных процедур, работающий поверх протокола HTTP/2. Как и во многих RPC-системах, gRPC основан на идее определения сервиса, указывающий методы, которые можно вызвать удаленно с их параметрами и возвращаемыми типами [8].

На стороне сервера реализуются методы, которые предоставляются для клиентов, и запускается gRPC-сервер для обработки клиентских запросов. На стороне клиента используется заглушка, которая предоставляет те же методы, что и сервер. Его высокая производительность достигается за счет использования протокола HTTP/2 и Protocol Buffers.

Однако у gRPC есть ряд недостатков, которые стоит учитывать при проектировании систем с применением данной технологии. Во-первых, Protocol Buffers это бинарный формат, поэтому для отладки и разработки приложений требуется иметь актуальную версию схемы данных, что делает невозможным отправку с клиента произвольных запросов и затрудняет отладки вне слоя приложения, где данные не представлены человекочитаемым форматом. Во-вторых, несмотря на то, что в 2022 была представлена спецификация протокола HTTP/3, HTTP/2 все еще имеет ограниченную поддержку со стороны браузеров, что делает невозможным непосредственную коммуникацию между веб-клиентом и сервисами.

Частично это проблема решается специальными расширениями, которые добавляют HTTP-прокси слой, который проксирует HTTP/1 запросы с небинарным форматом данных, например, JSON, в HTTP/2 запросы с данными в формате Protocol Buffers, однако это нивелирует достоинства бинарной сериализации из-за необходимости дополнительной конвертации данных между форматами.

Поэтому при проектировке систем общеиспользуемый REST API интерфейс присутствует лишь во внешнем слое приложения и может быть реализован без расширений, исходя из конкретной бизнес-логики, что добавляет накладные расходы на цикл сериализации и десериализации лишь во входном и выходном запросах.

2.3 Обзор языка программирования Golang

Go (также известный как Golang) является статически типизированным компилируемым языком программирования, разработанным на замену ранним языкам системного программирования, таким как C^{++} и Java. Созданный Google, Go предоставляет упрощенный синтаксис и мощные инструменты для разработки операционных систем, сетевых сервисов, микросервисов и других распределенных систем [9].

Для простоты внедрения данного языка было уделено большое внимание простоте синтаксиса и грамматике языка. Многие задачи имеют лишь одно решение средствами языка Go, которое описывается общеизвестными методами парадигм ООП и структурного программирования. За счет просты грамматики также достигается высокая скорость компиляции, что положительно сказывается на опыте разработки.

Также для устранения факторов, мешающих разработке, в стандартной поставке языка также предусмотрены инструменты для управления зависимостями и сборки, форматирования кода, синтаксического анализа, генерации и просмотра документации, а также профилирования готовых приложений.

Несмотря на то, что исходные коды языка находятся в общем доступе под свободной лицензией, процесс разработки по большей части контролируется разработчиками Google и многие решения проходят процесс согласования с проектировщиками языка в лице Кена Томпсона, который участвовал в разработке системы UNIX, Роба Пайка, известного за вклад в развитие операционной системы Plan9, и Роберта Гриземера, который до этого работал над виртуальной машиной V8 для языка JavaScript.

Поэтому, с одной стороны, участие сторонних разработчиков над ядром языка ограничено, поэтому новый функционал принимается с большим количеством обсуждений и согласований, за что язык регулярно критикуют.

Есть мнение, что в текущей реализации языка отрицается многолетний опыт разработки других языков, что отражается в использовании очень простой системы типов, сборщика мусора на поколенческом алгоритме и общей невыразительности базовых синтаксических элементов языка .

С другой стороны, подобный подход гарантирует полную обратную совместимость со старыми программами и маленькое ядро языка с выверенным набором инструментов, которые призваны уменьшать количество дискуссий

по поводу вопросов, которые не имеют к разработке прямого отношения: сборка, форматирование, дистрибуция. Что в связке с богатой библиотекой для работы с сетевыми стеками делает Golang хорошим выбором при разработке программ, связанных с веб-разработкой, и послужило причиной выбора данного языка в разработке программного обеспечения для данной работы.

2.4 Основные инструменты построения распределенных архитектур

Одним из ключевых вопросов при построении распределенных архитектур является асинхронная коммуникация между миросервисами, которая позволяет независимо масштабировать ее компоненты.

Брокер сообщений является критически важным компонентом при проектировании архитектуры микросервисов. Работая как посредник при передаче данных между различными сервисами, он позволяет минимизировать прямые связи и обеспечивает ряд ключевых преимуществ, которые облегчают разработку, масштабирование и поддержку сложных микросервисных систем.

Одно из основных преимуществ использования брокера сообщений - декуплинг различных компонентов системы. В архитектуре микросервисов декуплинг является важным фактором в обеспечении независимости каждого из сервисов. Благодаря брокеру сообщений, компоненты в системе не должны прямо общаться друг с другом, что повышает их изоляцию и позволяет им работать независимо. Это упрощает процесс обновления или модификации отдельных компонентов, поскольку минимизирует риск сбоев всей системы из-за изменений в одном из сервисов.

Далее, использование брокера сообщений увеличивает надежность системы. Благодаря использованию промежуточного слоя для передачи сообщений, они могут быть надежно сохранены и переданы даже при временных сбоях приема или трансляции. Это гарантирует, что важные данные не будут потеряны из-за сбоев или проблем с конкретными сервисами, усиливая отказоустойчивость всей системы.

Помимо этого, брокеры сообщений обеспечивают асинхронную передачу данных. Это означает, что система не требует немедленного ответа от сервиса-получателя, что может быть критически важно при высоких нагрузках. Брокеры сообщений могут агрегировать сообщения и управлять их доставкой оптимальным образом, обеспечивая более плавную и эффективную работу системы.

На рынке представлено множество различных брокеров сообщений, включая RabbitMQ, Apache Kafka, Google Pub/Sub и AWS Simple Queue Service (SQS). Каждый из этих продуктов предлагает различные функции и возможности, такие как надежные очереди сообщений, модели публикации/подписки, устойчивость к ошибкам и многое другое. Это позволяет разработчикам

выбирать наиболее подходящий брокер сообщений для их конкретных нужд и требований.

В итоге, брокеры сообщений играют ключевую роль в архитектуре микросервисов. Они повышают надежность и устойчивость системы, обеспечивают декуплинг между сервисами и позволяют более эффективно управлять данными и нагрузкой в рамках сложных микросервисных экосистем.

Amazon Simple Queue Service (SQS) является мощным инструментом для обработки сообщений в архитектуре микросервисов и обладает несколькими ключевыми преимуществами, которые делают его привлекательным выбором для различных сценариев использования.

Надежность: Amazon SQS предлагает высокую надежность благодаря инфраструктуре Amazon. Сервис гарантирует доставку сообщения хотя бы один раз, что помогает избежать потери данных. Кроме того, Amazon SQS предлагает очереди с повышенной отказоустойчивостью, которые хранят сообщения на протяжении определенного времени, пока они не будут успешно обработаны получателем.

Масштабируемость: Amazon SQS легко масштабируется, чтобы обеспечивать эффективную обработку больших объемов данных. Благодаря возможности бесконечного расширения, этот сервис может гибко настраиваться для поддержки микросервисов любого размера, чтобы поддерживать высокую производительность без дополнительных усилий с вашей стороны.

Безопасность: Благодаря тесной интеграции с AWS Identity and Access Management (IAM), Amazon SQS позволяет точно контролировать доступ к очередям сообщений, обеспечивая высокий уровень безопасности. Также доступны функции транзитного шифрования для дополнительной защиты данных.

Простота использования: Amazon предлагает удобные инструменты для работы с SQS, включая AWS Management Console, SDK и CLI. Это обеспечивает простой метод создания и управления очередями, отправки и приема сообщений и т. д.

Тонкая настройка и контроль: Amazon SQS позволяет настраивать множество параметров очереди, таких как максимальный размер сообщения, период видимости и период удержания, что позволяет оптимизировать сервис под конкретные потребности.

Интеграция с AWS: Если вы уже используете другие сервисы от AWS, такие как Lambda, S3 или EC2, использование SQS значительно упрощает

интеграцию микросервисов, поскольку все они работают вместе без каких-либо существенных проблем.

Таким образом, Amazon SQS представляет собой отличный выбор для обработки сообщений в архитектуре микросервисов благодаря своей надежности, масштабируемости, безопасности и простоте использования. Благодаря настройке под конкретные потребности и легкой интеграции с другими сервисами AWS, SQS может стать очень ценным активом в микросервисной архитектуре.

ГЛАВА 3

Распределенная микросервисная архитектура генерации изображений

3.1 Распределенная микросервисная архитектура генерации изображений

Исходя из перечисленных требований в предыдущей главе архитектура должна:

1. Предоставлять возможность горизонтального масштабирования каждой из частей системы по отдельности.

2. Предоставлять возможность инструментации для диагностики и сбора статистики.

3. Использовать сетевой транспорт, который обеспечивает отказоустойчивость и оптимальные тайминги сетевых запросов.

4. Позволять переиспользовать отдельные компоненты другим системам.

В рамках данной работы была реализована система для инференса изображений с HTTP API для внешних потребителей (рис. 3.1).

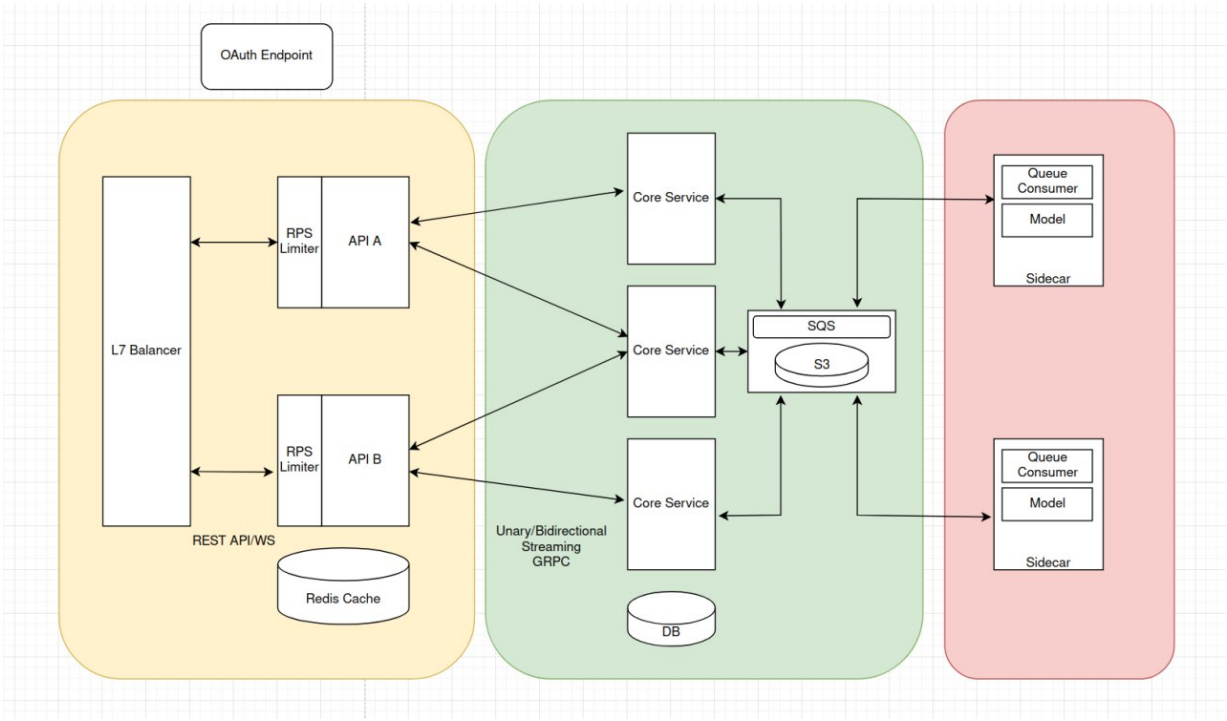

Рисунок 3.1 Схема архитектуры

Данная система состоит из трех основных частей:

1. HTTP Gateway для принятия внешних запросов.
2. Изолированный backend с холодным хранилищем пользовательских запросов генерации.
3. Сайдкар генерации с моделью генерации изображений с помощью нейронной сети.

HTTP Gateway, представленный желтой зоной на рисунке 3.1, предоставляет внешний слой приложения, который отвечает за бизнес-логику, работу с пользовательскими данными и идентификаторами и проксирует GRPC запросы походами в основной backend. Как правило, данная часть является наименее нагруженным компонентом системы, поэтому может быть представлена единичными экземплярами в каждой из локаций, известных L3 балансеру приложения.

Изоляция данного слоя также позволяет разделять источники траффика в систему и вводить разные правила на количество поступающих запросов и отдельно кэшировать ответы основного бэкенда.

На данном слое также представляется возможным размещение документации для разработчиков, которые работают с внешним API системы (рис. 3.2).

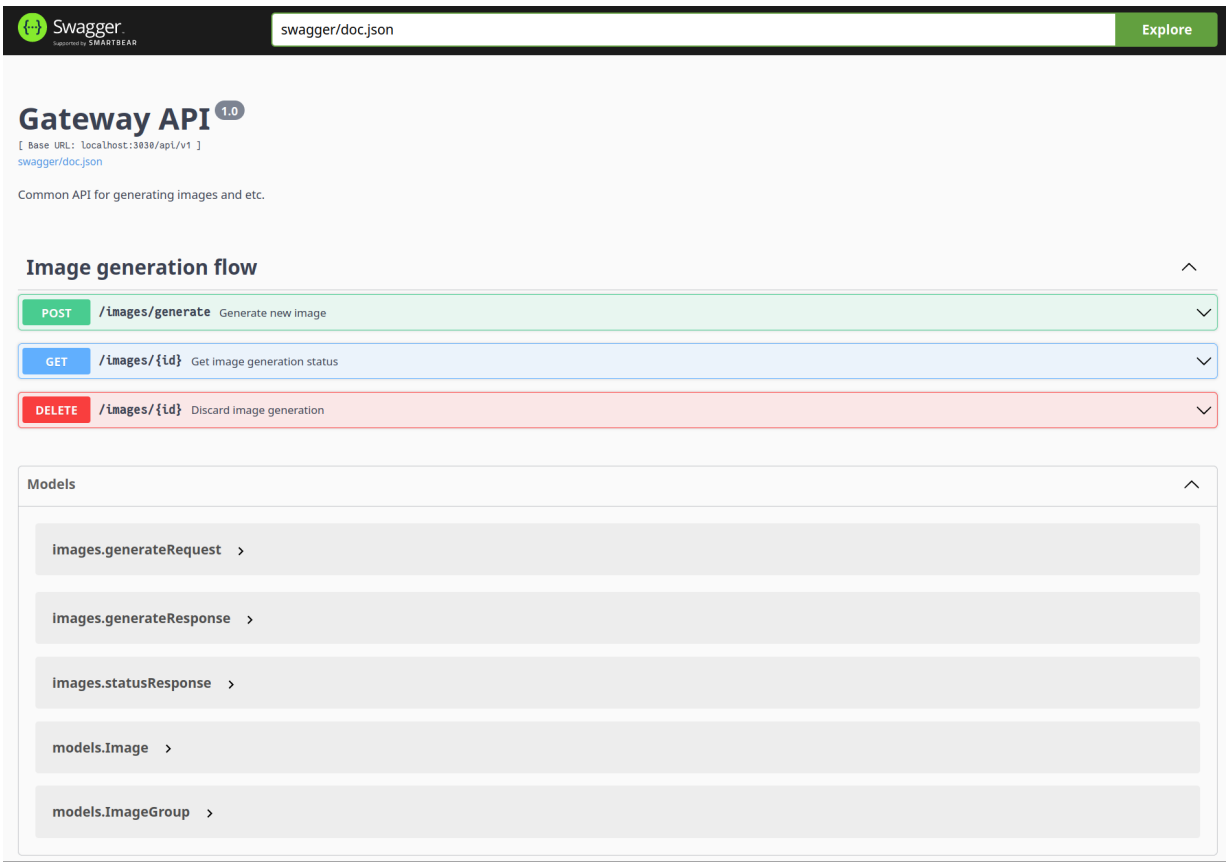

Рисунок 3.2 Внешняя документация эндпоинтов API

Итого, HTTP Gateway собирает необходимую информацию из пользовательского запроса и осуществляет сетевой поход по GRPC в основной backend с нужными параметрами.

Основной backend принимает запросы от HTTP-прокси и осуществляет все сетевые походы во внешние сервисы для их обслуживания. Для хранения информации об активных запросах, которые обрабатываются асинхронно, необходимо персистентное хранилище в виде, например, реляционной базы данных. В данном примере используется SQLite, так как эта база данных является наиболее простым в эксплуатации вариантом с возможностью последующей миграции на другие реляционные базы данных, типа PostgreSQL, MySQL и т.д. за счет минимальной спецификации своего диалекта SQL.

Помимо текстовых данных в этом узле системы необходимо работать с бинарными данными, например, изображениями, которые не рекомендуется хранить в реляционных базах данных. Одним из самых популярных вариантов хранения подобных данных является AWS S3 (Simple Storage Service). Делегирование хранения бинарных статичных данных в подобный сервис позволяет использовать CDN для кэширования наиболее востребованных файлов, что предоставляет оптимальное время загрузки данных на клиенте.

Внутренний backend приводит пользовательские HTTP запросы в версионируемые сообщения в Protobuf формате и записывает в SQS очередь, которую читает сайдкар генерации. Сайдкар генерации занимается непосредственным общением с моделью нейронной сети, которые зачастую требуют GPU ресурсы, которые менее доступны, чем CPU и RAM, поэтому количество сайдкаров можно масштабировать назависимо от основного backend-а по количеству доступных GPU.

При реализации взаимодействия между потребителем очереди, который может быть написан на платформе, отличной от рантайма инференса, возникают различные варианты реализации.

В случае наличия биндингов к рантайму инференса сайдкар может не производить межпроцессного взаимодействия и коммуницировать с моделью прямо в памяти (рис 3.3). Подобные варианты возможны при использовании общеизвестных рантаймов, типа TRT, Onnx и так далее, поскольку биндинги доступны для большого числа платформ. Однако это требует сохранения модели в установленном формате, что не всегда подходит для произвольной модели.

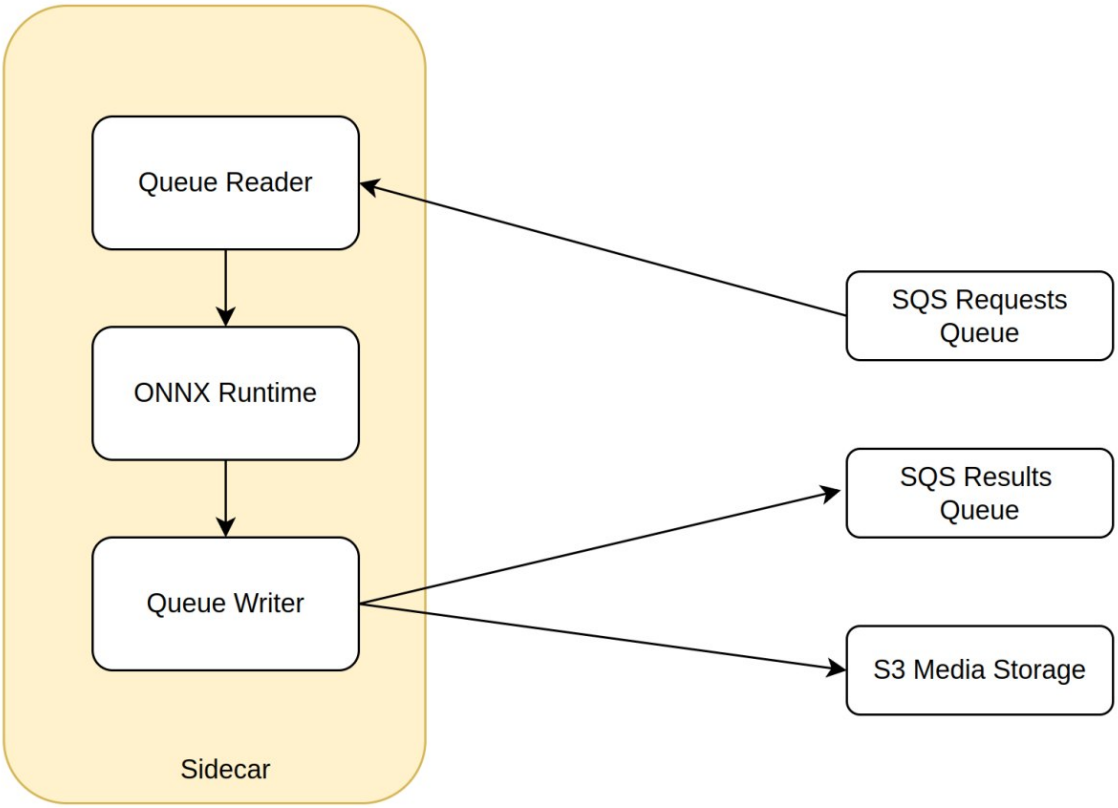

Рисунок 3.3 Нативный способ общения с моделью

В случаях, когда отсутствует реализация биндигов рантайма модели к

основному языку бэкенда, можно прибегнуть к сетевой передаче данных по HTTP (рис. 3.4). Достоинство данного метода заключается в простоте использования и внедрения. Данный подход не требует особого формата модели и может быть использован с верхнеуровневым кодом инференса. В то же время данный подход может не подойти в ситуациях, где предъявляются строгие требования к latency запросов, потому что передача крупных бинарных файлов медленнее передачи их по памяти процесса.

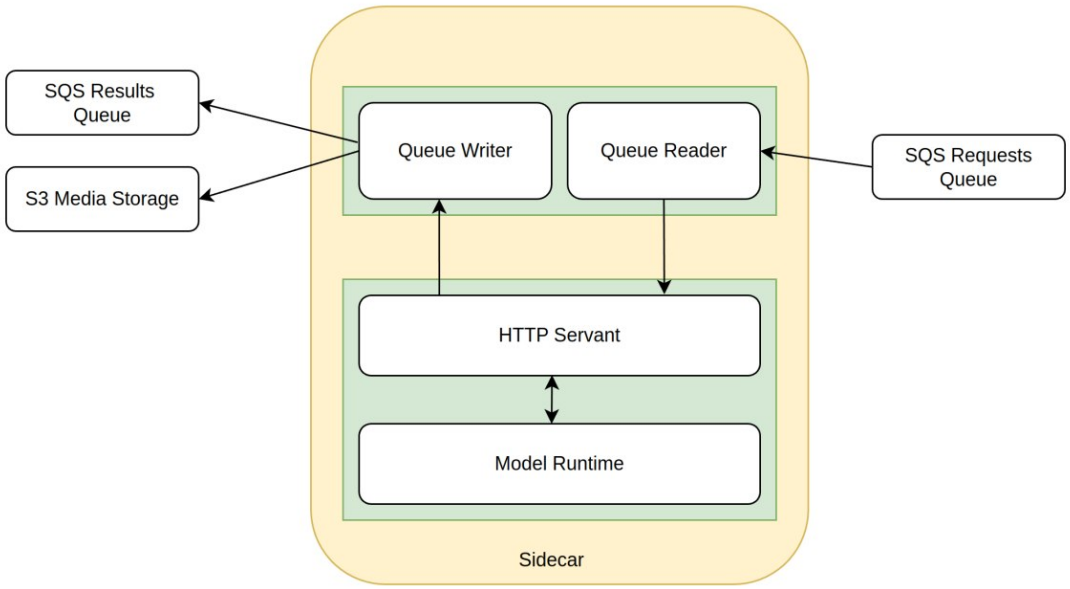

Рисунок 3.4 Общение по HTTP с моделью

Еще одним способом передачи данных по сети является общение по UDS (рис. 3.5), которое семантически похоже на общение по TCP с точностью до того, что транспорт данных осуществляется непосредственно через память ядра операционной системы. Данный способ показывает многократное улучшение в таймингах ответов по сравнению с передачей данных по HTTP, однако требует реализации сообственного бинарного протокола, что затрудняет разработку.

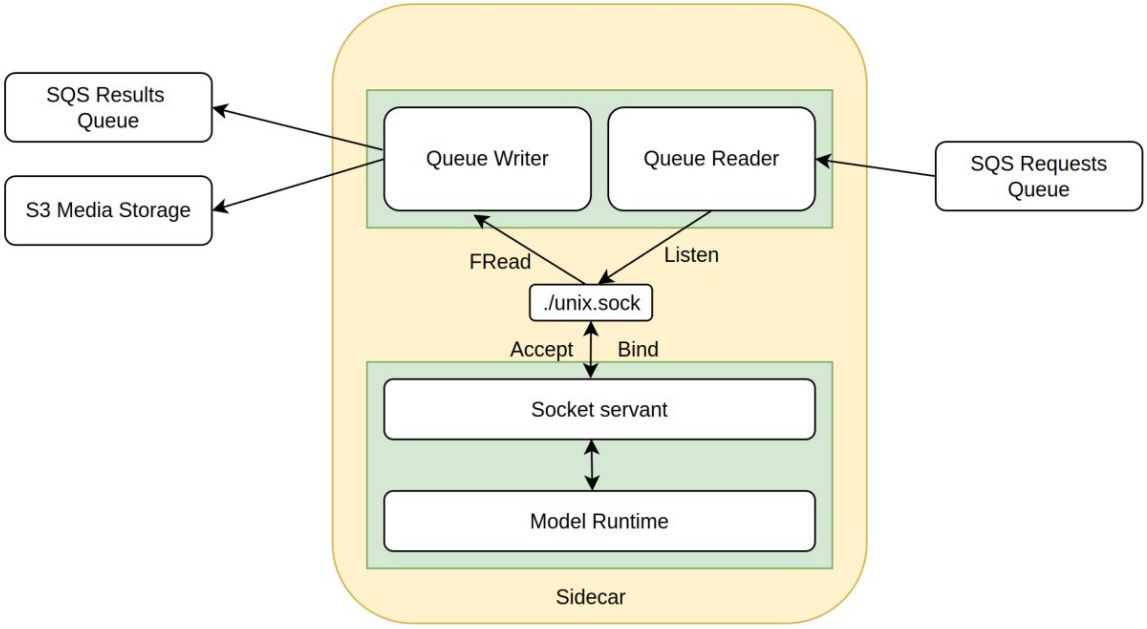

Рисунок 3.5 UDS общение с моделью

3.2 Сценарии использования системы генерации изображений

По умолчанию запрос на генерацию состоит из следующих шагов:

1. Клиент отправляет POST запрос с промптом и параметрами изображения.
2. HTTP Gateway собирает GRPC запрос и вызывает удаленную процедуру внутреннего backend-а.
3. Внутренний backend создает запись в реляционной базе данных и кладет запрос на генерацию в очередь сайдкара.
4. Сайдкар достает запрос из очереди и осуществляет инференс модели.
5. Полученное изображение перекладывается в S3, а ссылка вместе с инденитификаторами генерации отправляется в ответную очередь.
6. При поступлении ответного сообщения изображение перекладывается из временного бакета S3 в публичный, а у генерации в базе данных обновляется статус.

7. Клиент, получив идентификатор генерации, осуществляет поллинг хендлера статуса до подтверждения завершения, и в конце получает ссылку на полученнное изображение.

Гарантии на обработку сообщения осуществляются за счет Aknowledge семантики чтения сообщений из очереди. В SQS это реализовано через выставление visibility timeout, в течение которого сообщение не может быть заново прочитано другим потребителем. При успешной обработке сообщение удаляется из очереди. При неуспешной обработке сообщение снова становится видим для других потребителей и повторно обрабатывается до удаления из очереди.

При возникновении неопределенного поведения или возникновения неконсистентности данных статус может не обновиться. Для этого предлагается установить TTL на таблицы, связанные с хранением данных о генерации, для возможности проставления соответствующего HTTP статуса в API.

В случае, когда время обработки сообщений превышает интервал, с которым в систему поступают новые запросы, сообщения становятся в очередь, и время обработки сообщения будет включать не только время непосредственной обработки, но и время пребывания в очереди.

Для масштабирования поступающей нагрузки необходимо увеличивать количество сайдкаров, которые могут обрабатывать очередь независимо друг от друга. Так же уменьшение времени обработки сообщения можно достичь за счет батчевания поступающих сообщений: подобная обработка увеличивает время обработки каждого отдельного сообщения, но дает выигрыш для группы сообщений, что увеличивает общую пропускную способность системы.

Как было описано выше, при использовании батчинга можно обрабатывать несколько сообщений одновременно (рис. 3.6) Однако в этой схеме остаются блокирующие io операции, которые удерживают ресурс GPU во время сборки сообщения и отправки изображений в S3.

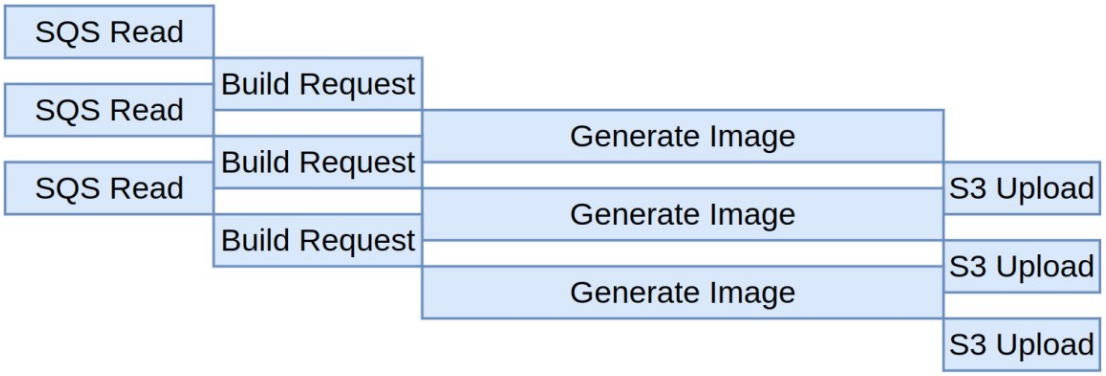

Рисунок 3.6 Батчевая обработка сообщений

Во избежание простоя GPU между отправкой сообщений можно захватывать сообщения наперед и отправлять их на генерацию во время отправки в S3 (рис. 3.7). Данная оптимизация может сэкономить до 200 мс при ожидании в очереди, что составляет около 5% от времени генерации изображения в разрешении 256x256 на видеокарте Nvidia Tesla V100.

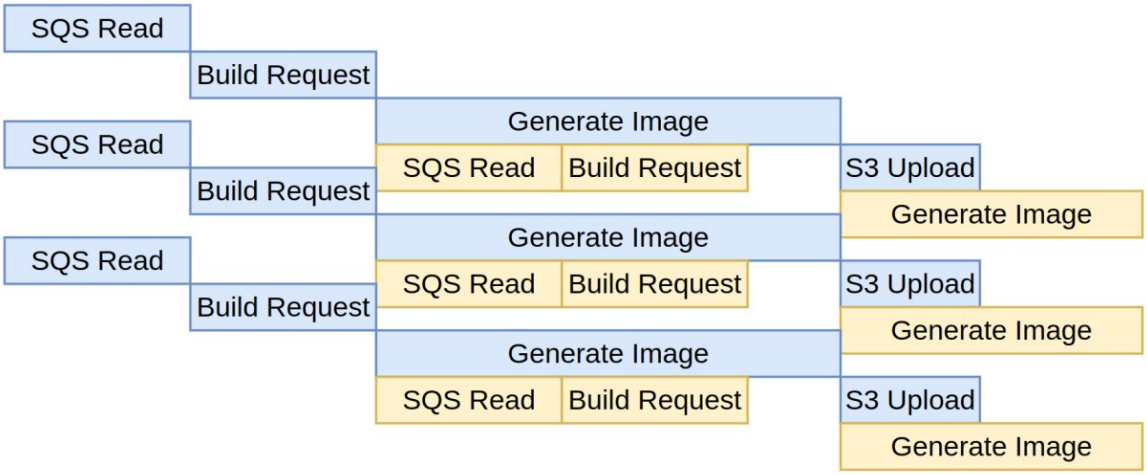

Рисунок 3.7 Батчевая обработка сообщений с захватом очереди

ЗАКЛЮЧЕНИЕ

Микросервисная методология позволяет решать ряд проблем связанных с масштабированием и отказоустойчивостью систем, однако налагает ряд ограничений и имеет ряд недостатков, с которыми приходится сталкиваться в процессе разработки. Некоторые из данных проблем предлагается решить путем переноса концепции наличия контракта формата передачи данных на взаимодействие между сервисами через введение понятия декларативной микросервисной архитектуры.

В ходе работы было рассмотрено понятие микросервисной архитектуры и произведен обзор имеющихся средств и методологий разработки, применяющихся для коммуникации веб-сервисов.

Результатом работы стала разработка программного обеспечения для генерации изображений с помощью нейронной сети с сетевым интерфейсом. Рассмотрены подходы общения разных процессов и проанализированы достоинства и недостатки каждого из методов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[2] К. Ричардсон. *Микросервисы: паттерны разработки и рефакторинга / Крис Ричардсон. - Санкт-Петербург [и др.] : Питер, Прогресс книга, 2020. - 542 с. - (Библиотека программиста).* Библиотека программиста. Питер Прогресс книга, Санкт-Петербург и др., 2020.

[3] M.L. Abbott and M.T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise.* Pearson Education, 2009.

[4] М. Клеппман. *Высоконагруженные приложения: программирование, масштабирование, поддержка: [перевод с английского] / Мартин Клеппман. - Санкт-Петербург [и др.] : Питер, Прогресс книга, 2018. - 637 с. - (Бестселлеры O'Reilly).* Бестселлеры O'Reilly. Питер Прогресс книга, Санкт-Петербург и др., 2018.

[5] Marek Bolanowski, Kamil Żak, Andrzej Paszkiewicz, Maria Ganzha, Marcin Paprzycki, Piotr Sowiński, Ignacio Lacalle, and Carlos E. Palau. *Eficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems.* IOS Press, September 2022.

[6] Binary formats shootout [Электронный ресурс] / bradlee speice. - Режим доступа: https://speice.io/2019/09/binary-format-shootout.html. - Дата доступа 10.12.2023.

[7] Protocol buffers [Электронный ресурс] - Режим доступа: https://protobuf.dev. - Дата доступа 10.12.2023.

[8] grpc [Электронный ресурс] - Режим доступа: https://grpc.io. - Дата доступа 10.12.2023.

[9] Golang [Элекронный ресурс] - Режим доступа: https://go.dev/project. - Дата доступа 10.12.2023.

ПРИЛОЖЕНИЕ А

Репозиторий приложения на GitHub

Код приложения находится в открытом доступе на платформе коллаборации GitHub по ссылке, представленной в виде QR-кода. Практическая часть, упоминаемая в основной работе, представлена пакетами cmd и internal. Также присутствует docker-compose файлы необходимые для поднятия нужного окружения с томами для брокера сообщений, базы данных, хранилища медиа-файлов и инструментации приложения.