ВВЕДЕНИЕ

Мир информационных технологий развивается с невероятной скоростью. Это подталкивает к необходимости поддерживать высокие стандарты качества программ и дает возможности для их масштабирования. Важной концепцией в этом контексте является контейнеризация. Она позволяет упаковывать приложения в стандартизированные контейнеры. Эти контейнеры содержат весь необходимый код и компоненты для работы приложения. Таким образом, разработчики могут легко создавать и распространять свои программы.

Целью данного дипломного проекта является разработка и реализация системы управления контейнеризированными приложениями в распределенной среде. По сути, это проект о создании инструмента для управления процессами развертывания, масштабирования и обслуживания контейнеров. Проект нацелен на повышение гибкости использования контейнеров, оптимизацию ресурсов и повышение общей эффективности операций. Ключевые функции включают автоматическое масштабирование в ответ на изменения нагрузки, балансировку нагрузки для равномерного распределения трафика и интегрированный мониторинг состояния системы. Эти возможности предоставят пользователям удобный инструмент для управления приложениями. В рамках проекта был проведен тщательный анализ и применение последних достижений в области контейнеризации и микросервисной архитектуры. Особое внимание было уделено интеграции с Docker, а также использованию инструментов для эффективной оркестровки контейнеров, что позволило создать мощную и гибкую основу для будущей системы. Проект основан на языке Python, выбор которого обусловлен его гибкостью и широким спектром библиотек, специализированных под задачи работы с Docker API. Это дало возможность максимально эффективно взаимодействовать с контейнерами и управлять ими.

Результатом работы является разработка платформы управления контейнерами, которая отличается простотой и удобством использования. Разработанная система предлагает решения для облачных вычислений, DevOps и Continuous Integration / Continuous Deployment (CI/CD), ускоряя разработку и доставку продуктов на рынок. Проект упрощает разработку программного обеспечения, делая её доступнее.

Интеграция современных подходов к мониторингу и аналитике позволяет оперативно реагировать на изменения в работе приложений и поддерживать их стабильную и эффективную работу.

В процессе работы над проектом было решено достаточно много задач, связанных с автоматическим масштабированием, сбором метрик работы контейнеров и оптимизацией распределения ресурсов между ними. Всё это в совокупности представляет значительный вклад в развитие инструментов управления программными приложениями в IT.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Введение в контейнеризацию и системы управления

Контейнеризация представляет собой метод виртуализации на уровне операционной системы, который позволяет запускать и управлять приложениями и их зависимостями в процессах, изолированных от остальной системы. Этот подход позволяет легко развертывать и масштабировать приложения в любом окружении, поддерживая при этом их работоспособность и совместимость. Наглядная демонстрация такого подхода представлена на рисунке 1.1, которая иллюстрирует концепцию контейнеризации, показывая, как приложения, упакованные в контейнеры, изолируются друг от друга и одновременно работают на одной и той же хост-системе.

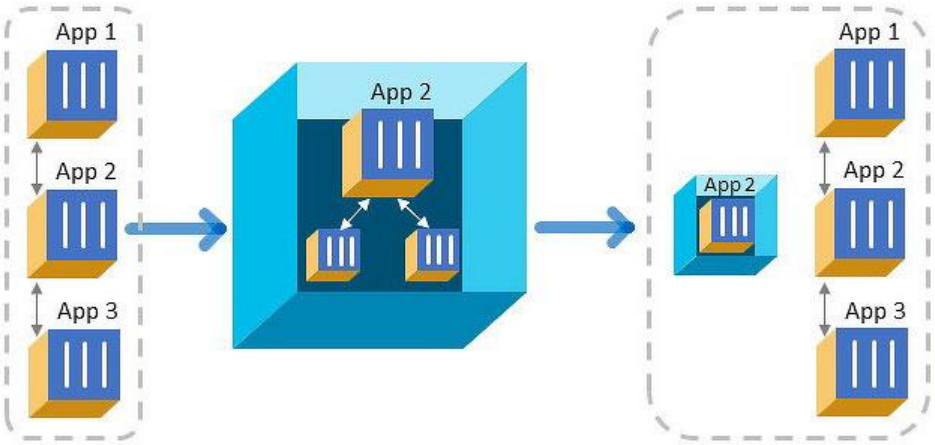

Рисунок 1.1 – Схема контейнеризации

История Docker начинается с презентации Соломона Хайкса на PyCon в 2013 году, что стало значимым моментом в разработке программного обеспечения для приложений. Docker Inc. была основана Камелем Фунаиди, Соломоном Хайксом и Себастьеном Палем в рамках летней программы Y Combinator 2010 года, выросшая из dotCloud – компании, предоставляющей платформу как услугу. Эта компания стала истоком для Docker, который был выпущен как открытый проект в марте 2013 года, переходя от использования LXC к собственному компоненту, libcontainer, что улучшило независимость и эффективность программного обеспечении [1].

За годы своего развития Docker расширился за пределы своих первоначальных Linux x86 основ, поддерживая другие операционные системы и архитектуры, включая Windows и Arm. Это расширение, вместе с внедрением оркестраторов, таких как Kubernetes от Google, значительно

увеличило влияние Docker, облегчая разработку, обмен и развертывание приложений с невиданной ранее скоростью и эффективностью. Сообщество Docker значительно выросло, Docker Hub и Docker Desktop обслуживают миллионы активных пользователей.

Путь Хайкса, от его ранних дней программирования до управления Docker как смесью открытого исходного кода и бизнеса, подчеркивает трансформационную силу контейнеризации в IT-ландшафте. Успех Docker является примером безупречного сочетания технологии и предпринимательства. Эта философия продвинула Docker в качестве ведущей силы в технологическом мире, стимулируя движение контейнеризации и влияя на траекторию современной разработки программного обеспечения. С развитием области контейнеризации появилась необходимость в их оркестрации, что представляет собой автоматизированное развертывание, масштабирование и управление контейнерными приложениями. Kubernetes, разработанный Google, стал одним из ключевых инструментов в этом процессе, обеспечивая не только распределение нагрузки между контейнерами, но и их взаимодействие в рамках микросервисной архитектуры. Это значительно упрощает разработку и поддержку сложных приложений, делая их более надежными и масштабируемыми [2].

Инструменты вроде Docker и Kubernetes легли в основу современного программного обеспечения и IT-инфраструктур. Внедрение контейнеризации способствует ускорению процессов разработки, тестирования и развертывания, повышает надежность приложений и упрощает их масштабирование. Одним из важных аспектов контейнеризации также является улучшение безопасности, так как каждый контейнер функционирует как изолированное окружение, снижая тем самым риски вторжения и распространения вредоносного код.

Научное сообщество активно исследует эти технологии, что отражается в большом количестве академических работ и публикаций. В рамках этих исследований выявляются новые методы увеличения эффективности контейнеров, разрабатываются рекомендации по их оптимизации и эксплуатации, а также изучается взаимодействие контейнеризированных приложений с облачными сервисами и инфраструктурой.

Так, контейнеризация и системы оркестрации контейнеров стали неотъемлемой частью современной ИТ-индустрии, предоставляя разработчикам и инженерам эффективные инструменты для создания и поддержания сложных и масштабируемых приложений.

1.2 Анализ существующих систем управления контейнерами

1.2.1 Kubernetes (K8s)

Kubernetes, часто сокращаемый до K8s, является мощным

инструментом для автоматизации развертывания, масштабирования и управления контейнеризированными приложениями в различных средах, от облачных сервисов до частных и гибридных облаков. Созданный инженерами Google на основе их обширного опыта работы с контейнерами и поддерживаемый Cloud Native Computing Foundation, Kubernetes быстро стал стандартом в области оркестрации контейнеров [3].

Сердцем Kubernetes является автоматизация развертывания приложений. Ключевой особенностью является способность не только упростить развертывание приложений в контейнерах, но и предоставить инструменты для их непрерывного управления и масштабирования. Это достигается за счет использования мощной абстракции, которая описывает желаемое состояние приложений и их окружения, а затем автоматически изменяет реальное состояние, чтобы соответствовать заданному.

Система автоматического масштабирования Kubernetes позволяет приложениям реагировать на изменения нагрузки путем добавления или удаления ресурсов. Это значит, что при увеличении нагрузки на приложение, система может автоматически добавить дополнительные контейнеры для обработки дополнительных запросов, а затем также автоматически их убрать, когда нагрузка снижается.

Сервисное обнаружение и балансировка нагрузки в Kubernetes обеспечивают, чтобы входящий трафик распределялся между контейнерами. Это улучшает отказоустойчивость и общую производительность приложений. Управление конфигурациями и секретами помогает обеспечить безопасность приложений и данных, позволяя централизованно управлять конфигурационными данными и паролями, сертификатами или токенами, необходимыми для доступа к различным внешним ресурсам.

Кроме того, Kubernetes предлагает возможности по самоисцелению приложений, такие как автоматический перезапуск контейнеров, которые вышли из строя, замену и репликацию контейнеров, а также перераспределение ресурсов в случае выхода из строя узла.

Концепция кластера в Kubernetes строится на концепци, являющегося набором узлов, которые обеспечивают запуск контейнерных приложений. В контексте дипломного проекта, кластер Kubernetes будет изучаться как центральный элемент системы управления, способный обеспечивать высокую доступность и отказоустойчивость развертываемых приложений.

Структурно кластер Kubernetes состоит из главных (Master Nodes) и рабочих узлов (Worker Nodes), где главные узлы отвечают за координацию кластера и принятие решений о запуске и распределении приложений, а рабочие узлы – за непосредственное выполнение задач.

К основным компонентам рабочего узла можно отнести: Kubelet – это агент, работающий на каждом рабочем узле, отвечающий за запуск, остановку и поддержание работоспособности контейнеров в соответствии с указаниями API сервера, Kube-Proxy поддерживает сетевые правила на узлах, позволяя сетевому взаимодействию между контейнерными подами,

Container Runtime отвечает за запуск контейнеров, используя Docker, containerd, CRI-O или любой другой совместимый с CRI (Container Runtime Interface).

Поды и сервисы: контейнеры группируются в поды – основные развертываемые единицы в Kubernetes, которые могут содержать один или несколько тесно связанных контейнеров. Поды в свою очередь управляются сервисами, предоставляющими постоянные точки доступа к наборам работающих подов.

Касаемо архитектуры Kubernetes следует сказать следующее, работающий кластер Kubernetes включает в себя агента, запущенного на нодах (kubelet) и компоненты мастера (APIs, scheduler, etc), поверх решения с распределённым хранилищем. Приведённая схема на рисунке 1.2 показывает желаемое, в конечном итоге, состояние.

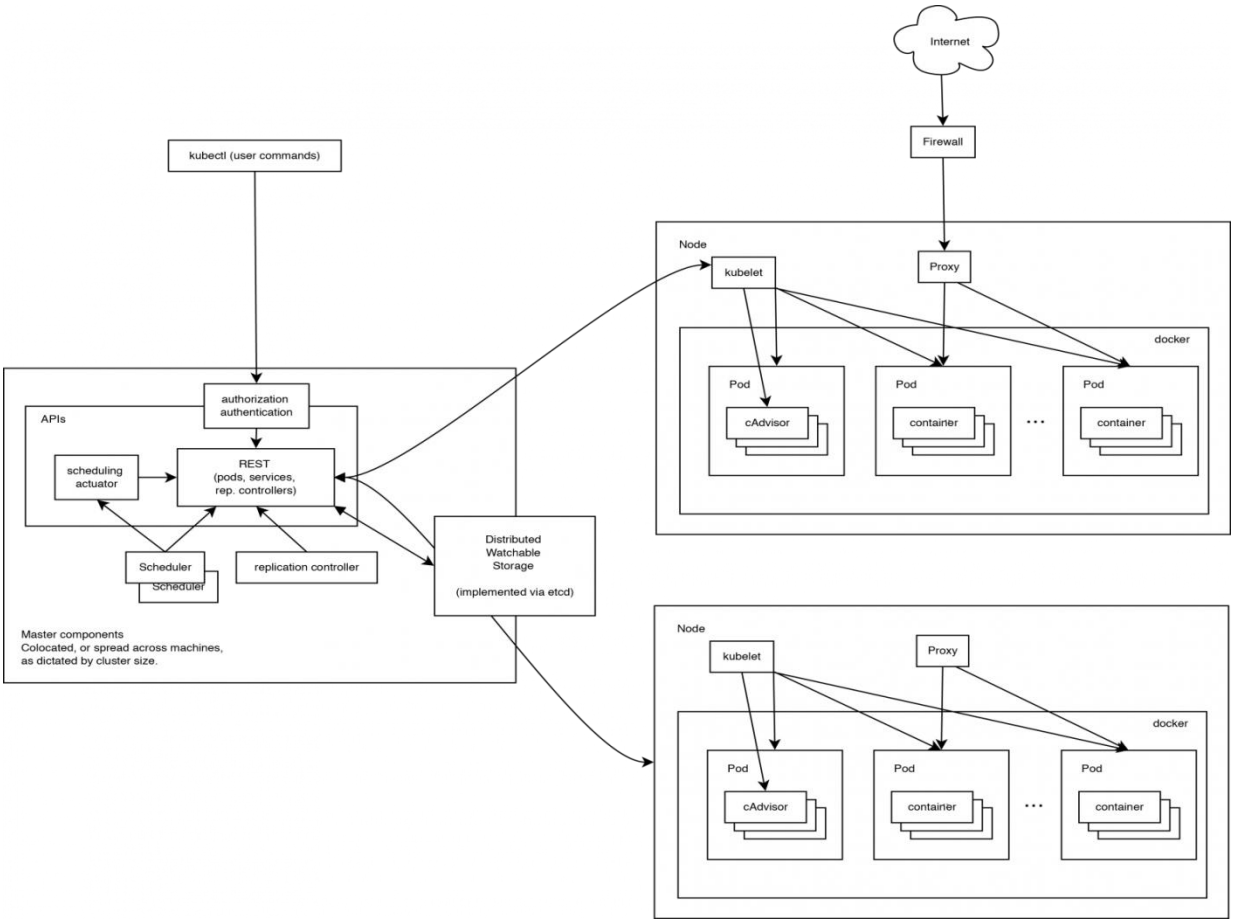

Рисунок 1.2 – Схема состояний Kubernetes

Благодаря своим расширенным возможностям, Kubernetes стал важным компонентом в стратегиях DevOps и CI/CD, облегчая непрерывную интеграцию и непрерывное развертывание программного обеспечения. Это позволяет организациям быстрее и более надежно выводить на рынок новые версии приложений, улучшая тем самым их конкурентоспособность [4].

1.2.2 Docker Swarm

Управление контейнерами становится более сложным по мере увеличения их количества, а системы оркестрации, такие как Docker Swarm, предлагают решения для этой проблемы. Docker Swarm, встроенный в Docker Engine начиная с версии 1.12, объединяет несколько хостов Docker в единый кластер, упрощая развертывание и масштабирование контейнеров. Это делает Swarm идеальным для начинающих в области оркестрации контейнеров благодаря его простоте установки и управления [5].

Архитектурно Docker Swarm организован вокруг узлов управления и рабочих узлов. Узлы управления координируют кластер, принимая решения о распределении задач, в то время как рабочие узлы выполняют эти задачи, запуская контейнеры с приложениями. Архитектура Docker Swarm представлена на рисунке 1.3.

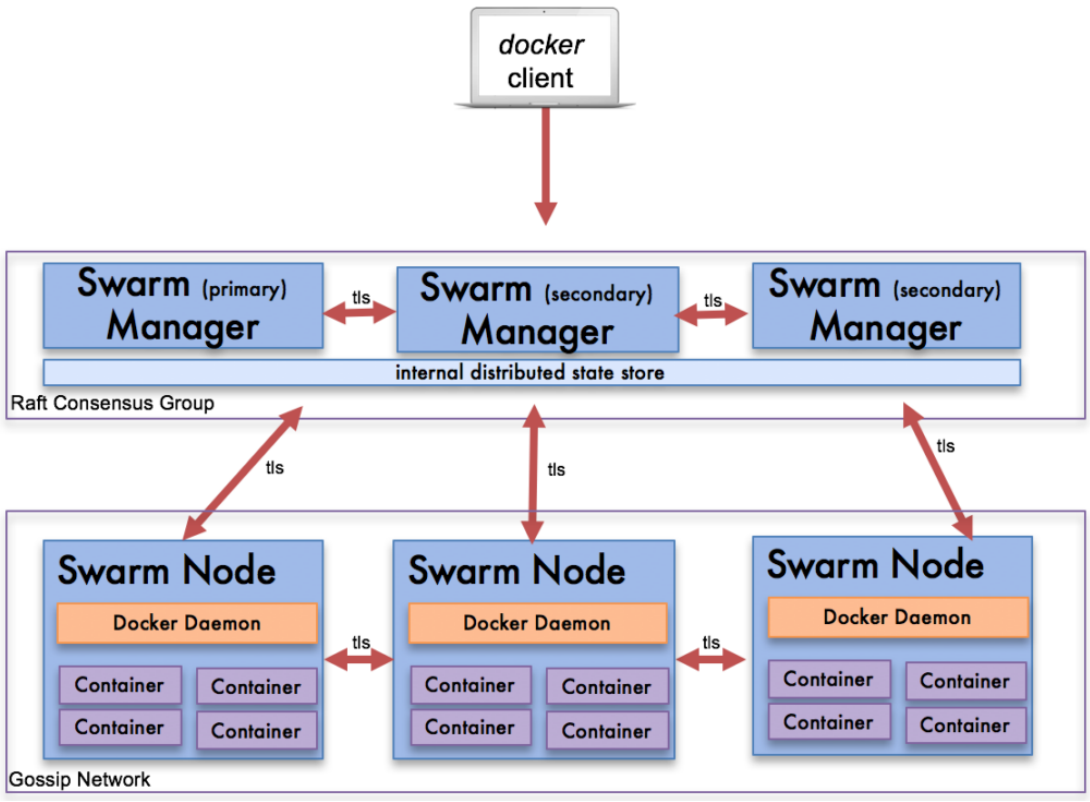

Рисунок 1.3 – Архитектура Docker Swarm

Хотя Docker Swarm отличается простотой и легкостью интеграции, его функционал может быть ограничен по сравнению с более масштабируемым и функциональным Kubernetes, что делает Swarm более подходящим для проектов меньшего масштаба.

Сравнивая Docker Swarm и Kubernetes, стоит отметить, что K8s предлагает значительно более расширенный набор функций и возможностей, что сопровождается повышенной сложностью установки и настройки. Тем не

менее, этот оркестратор является предпочтительным выбором для крупномасштабных и сложных проектов, требующих автоматического масштабирования и управления.

Docker Swarm, с другой стороны, хоть и уступает в возможностях, но его простота и интеграция непосредственно с Docker делают его удобным для меньших по размеру проектов, где не требуются продвинутые функции управления.

1.2.3 Apache Mesos

Apache Mesos представляет собой высокоуровневую абстракцию над кластерами машин, позволяя эффективно управлять ресурсами и распределять задачи среди большого количества серверов. Он предоставляет разработчикам программный интерфейс для управления ресурсами, что делает его особенно полезным в условиях распределённых вычислений и работы с Big Data [6].

Основная концепция, лежащая в фундаменте Mesos, заключается в его способности делегировать управление запуском задач двухуровневому планировщику. Это означает, что Mesos выступает в качестве "мастера", координирующего распределение ресурсов, в то время как фреймворки, такие как Marathon или Chronos, непосредственно управляют запуском приложений, учитывая предложенные ресурсы. На рисунке 1.4 изображена архитектура кластерного менеджера Apache Mesos.

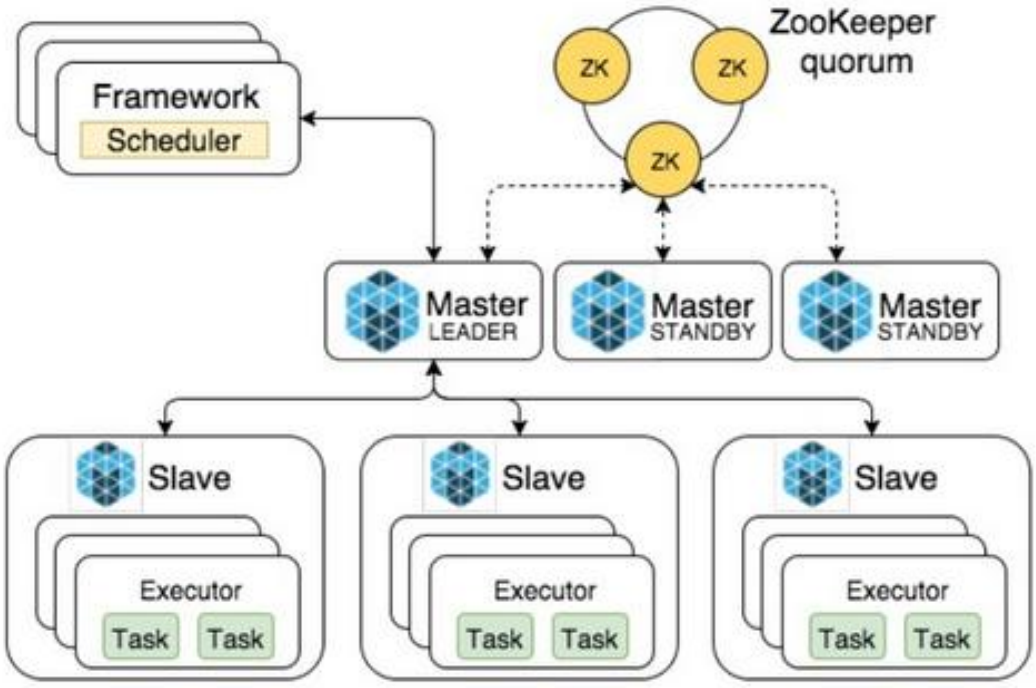

Рисунок 1.4 – Архитектура Apache Mesos

В верхней части рисунка 1.4 находится планировщик фреймворка

(Framework Scheduler), который взаимодействует с Mesos через ZooKeeper. ZooKeeper – это централизованная служба для обслуживания распределённых систем, которая используется для согласования кластера. Также на рисунке 1.4 показано, что планировщик фреймворка подключается к ZooKeeper, который состоит из трех узлов (ZK), обеспечивая отказоустойчивость и согласованность данных [7].

Ниже ZooKeeper располагаются узлы Mesos: один мастер-узел (Master Leader), который активен, и два мастер-узла в режиме ожидания (Master Standby), готовые взять на себя управление в случае сбоев.

Ещё ниже расположены рабочие узлы (Slave), которые выполняют задачи (Task). Каждый рабочий узел содержит исполнителя (Executor), который непосредственно запускает и управляет задачами. Executor может выполнять несколько задач одновременно. Эта модель позволяет Mesos эффективно распределять ресурсы и задачи по кластеру, повышая его производительность и масштабируемость.

Apache Mesos поддерживает контейнеризацию через Mesos Containerizer и интеграцию с Docker, что позволяет разработчикам запускать контейнеры в распределенных средах и масштабировать их согласно потребностям проекта. Это обеспечивает гибкость при работе с контейнеризированными приложениями, а также упрощает процесс масштабирования и управления приложениями в кластере.

Apache Mesos подходит для крупных развертываний, где требуется глубокий уровень контроля над ресурсами и где задачи имеют разнообразные требования к вычислительным ресурсам. Он широко используется в индустрии, в частности, для управления ресурсами в облачных средах и центрах обработки данных, где требуется высокая степень масштабируемости и надежности.

С одной стороны, преимуществом Mesos является его мощная и гибкая модель управления ресурсами, что делает его идеальным для запуска и управления масштабируемыми приложениями. С другой стороны, высокий порог входа и сложность управления может оказаться барьером для новых пользователей и проектов с меньшими требованиями к инфраструктуре.

В целом, Apache Mesos представляет собой надежный и мощный инструмент для управления ресурсами в крупномасштабных вычислительных средах, способный обеспечить эффективное выполнение задач в разнообразных доменах, включая облачные вычисления, обработку данных и микросервисные архитектуры.

В отличие от Mesos, Kubernetes предлагает более широкий спектр возможностей для оркестрации контейнеризированных приложений, облегчая развертывание, масштабирование и управление, что делает его предпочтительным для общих случаев использования. Docker Swarm же выделяется своей простотой и удобством в управлении контейнерами для меньших или менее сложных проектов.

1.3 Практическое применение контейнеризации

Приложения и сервисы в современной информационной среде обязаны быть адаптируемыми к быстрым изменениям, обеспечивать безотказную работу и быть готовыми к масштабированию в соответствии с потребностями пользователей. Технология контейнеризации отвечает этим требованиям и предлагает средства для оптимизации всего жизненного цикла приложений. Ниже описаны ключевые аспекты практического применения контейнеризации, актуальные для сферы информационных технологий и за пределами её.

Эффективность разработки и эксплуатации: Контейнеры гарантируют однородность среды на всех этапах разработки, тестирования и развертывания, что упрощает интеграцию и непрерывную доставку (CI/CD) и сокращает «время до рынка» (time-to-market) для новых версий приложений.

Гибкость масштабирования: Системы управления контейнерами, такие как Kubernetes, предоставляют инструменты для автоматического масштабирования приложений в ответ на изменение нагрузки, тем самым оптимизируя использование ресурсов и обеспечивая требуемый уровень сервиса.

Безопасность и изоляция: Контейнеры изолируют приложения друг от друга, уменьшая риск системных уязвимостей. Изоляция также способствует соблюдению стандартов безопасности и соответствия регуляторным требованиям.

Поддержка микросервисной архитектуры: микросервисы часто реализуются с помощью контейнеризации, что позволяет разрабатывать и развертывать независимые компоненты сложных приложений, ускоряя обновления и улучшения.

Портативность приложений: контейнеры обеспечивают независимость приложений от инфраструктуры, что позволяет переносить их между локальными серверами, частными и общедоступными облачными средами без изменения кода.

Оптимизация затрат: контейнеризация позволяет уменьшить затраты на инфраструктуру за счёт повышенной плотности развертывания и улучшенного управления ресурсами, снижая общую стоимость владения (ОСВ) [8].

Модернизация и интеграция: технология контейнеризации способствует модернизации существующих приложений и их интеграции с современными облачными сервисами, а также облегчает миграцию легаси-систем.

В заключение, применение контейнеризации в индустрии ИТ не только способствует повышению эффективности разработки и эксплуатации приложений, но и оказывает весомое влияние на ускорение цифровой трансформации предприятий. Оно является критически важным элементом в стратегии повышения агильности бизнеса, обеспечивая необходимую

гибкость, скорость и стабильность в изменяющемся цифровом ландшафте.

1.4 Современные исследования и разработки

Современные исследования и разработки в области контейнеризации акцентируют внимание на интеграции с облачными хранилищами, обеспечивая масштабируемость и упрощение процессов обновления и резервного копирования приложений. Использование образов контейнеров позволяет ускорить развертывание новых версий приложений и гарантирует их непрерывную работу. Кроме того, облачные хранилища обладают рядом преимуществ, включая гибкость, доступность, безопасность и удобство использования, а также возможность синхронизации данных между устройствами.

Принципы работы облачных хранилищ и контейнеризации способствуют оптимизации и эффективности развертывания и масштабирования приложений, предоставляя решения вроде Docker и Kubernetes для автоматизации этих процессов. Это способствует не только повышению производительности и надежности, но и облегчает передачу и развертывание приложений на различных платформах.

С другой стороны, концепция нативной облачной архитектуры подчеркивает значимость надежности, самовосстановления, масштабируемости и экономичности, а также обеспечивает простоту сопровождения и повышенный уровень безопасности. Облачные решения предоставляют возможности для автоматизированного масштабирования приложений и оптимизации использования инфраструктурных ресурсов, что способствует снижению расходов и ускорению процессов разработки и отгрузки приложений. Подход cloud-native также предлагает независимость от конкретных вендоров, давая возможность запуска приложений в разнообразных облачных средах.

1.5 Выбор фреймворка для разработки

Выбор подходящего фреймворка и архитектурного паттерна для разработки дипломного проекта является ключевым решением, которое определяет не только производительность и масштабируемость конечного продукта, но и его гибкость в долгосрочной перспективе. В этом контексте, интеграция микросервисной архитектуры с использованием FastAPI в качестве основного фреймворка для разработки, а также Docker для контейнеризации приложений, представляет собой передовой подход, поддерживаемый современной средой разработки PyCharm. Эта интеграция не только обеспечивает мощную платформу для создания высокопроизводительных веб-приложений и сервисов, но также значительно упрощает процесс разработки и обеспечивает удобство поддержки проекта [9].

Архитектурный паттерн микросервисов выбран за его гибкость, масштабируемость и возможность независимой разработки и развертывания каждого сервиса. Принципы микросервисов позволяют разделять функционал проекта на мелкие, легко управляемые части, что упрощает тестирование, поддержку и развитие каждого компонента системы. Для упрощения развертывания и обеспечения согласованности окружений на всех этапах жизненного цикла приложения используется Docker, обеспечивающий контейнеризацию сервисов.

FastAPI, зарекомендовавший себя как передовой и высокопроизводительный веб-фреймворк для создания API в экосистеме Python 3.6 и выше, выделяется среди прочих решений благодаря своему уникальному подходу к разработке. Он применяет стандартные типы Python для объявления переменных, что позволяет разработчикам пользоваться мощью статической типизации, обеспечивая при этом автоматическую валидацию данных и сериализацию и десериализацию без необходимости писать обширный код для этих целей.

Одним из ключевых преимуществ FastAPI является его выдающаяся производительность. Благодаря асинхронной поддержке, веб-фреймворк способен обрабатывать огромное количество запросов параллельно, не жертвуя при этом скоростью ответа. Это ставит FastAPI в один ряд с самыми производительными веб-фреймворками на Python, делая его особенно привлекательным для разработки высоконагруженных приложений и сервисов.

Асинхронная поддержка в FastAPI не просто является дополнительной функцией – она лежит в основе его дизайна. Это позволяет разработчикам эффективно использовать асинхронные запросы и обрабатывать I/O-операции, такие как обращения к базам данных или вызовы внешних API, без блокировки основного потока выполнения. Такой подход максимизирует производительность и эффективность приложений, разработанных на FastAPI.

В качестве среды разработки (IDE) используется PyCharm, которая предоставляет расширенную поддержку для Python и его фреймворков, включая FastAPI, упрощает работу с Docker и облегчает разработку микросервисов благодаря интегрированным инструментам для дебаггинга, тестирования и развертывания приложений. PyCharm также поддерживает виртуальные окружения, что позволяет легко управлять зависимостями проекта и обеспечивать его портативность [10].

Таким образом, интеграция микросервисной архитектуры с FastAPI в контейнерах Docker, разрабатываемых в PyCharm, обеспечивает эффективную и гибкую платформу для разработки современных веб-приложений и микросервисов, соответствующих текущим требованиям к производительности, масштабируемости и удобству разработки.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Изучив предметную область разрабатываемой системы, были разработаны основные требования, которые должны быть выполнены при реализации дипломного проекта. Для упрощения разработки системы разобьем ее на структурные блоки.

2.1 Описание основных блоков устройства

Для данного дипломного проекта были определены следующие блоки:

- блок интерфейса и взаимодействия пользователя;
- блок управления контейнерами;
- блок балансировщик нагрузки;
- блок контейнеров;
- блок операций масштабирования;
- блок анализа нагрузки;
- блок метрик;
- блок мониторинга.

Структурная схема, иллюстрирующая перечисленные блоки и связи между ними приведена на чертеже ГУИР.400201.071 С1.

Блоки были выделены таким образом, чтобы каждый из них выполнял определённую задачу, и обеспечивали корректную работу системы в целом. Далее рассмотрим принцип работы и задачи каждого из перечисленных выше блоков, а также их взаимодействие между собой.

2.2 Блок интерфейса и взаимодействия пользователя

Пользовательский интерфейс (UI) и программный интерфейс приложения (API) являются точкой входа для пользователей в систему управления контейнерами. Этот блок позволяет пользователям взаимодействовать с системой, выполнять запросы и получать результаты этих запросов. UI предоставляет графический интерфейс, который можно использовать для наблюдения за состоянием системы, управления контейнерами и конфигурации параметров, в то время как API предоставляет программные хуки для автоматизации этих процессов и интеграции с внешними системами.

Интеграция с блоком операций масштабирования подразумевает возможность отправки команд на изменение числа рабочих экземпляров контейнеров в зависимости от текущей нагрузки, полученной из блока метрик или блока мониторинга. Это обеспечивает гибкость системы, позволяя масштабировать ресурсы под текущие требования нагрузки и оперативно реагировать на изменения во внешней среде или в рамках самой системы.

Блок интерфейса и взаимодействия пользователя является ключевым

компонентом для обеспечения пользовательской интеграции с системой, представляя сложную инфраструктуру контейнеризации в удобной, интуитивно понятной и легкодоступной форме. Это центральное звено в обеспечении пользовательской адаптивности и оперативного управления приложениями и сервисами в динамически масштабируемой и многопользовательской среде.

2.3 Блок управления контейнерами

Блок управления контейнерами в системе управления контейнеризированными приложениями является ключевым элементом, отвечающим за координацию и управление контейнерами. Основная функция этого блока – оркестрация контейнеров, то есть управление их жизненным циклом, что включает в себя развертывание, мониторинг, масштабирование и обработку сбоев. Через программный интерфейс блок управления контейнерами автоматизирует развертывание контейнеров, регулирует их масштабирование в зависимости от изменения нагрузки и поддерживает оптимальное количество работающих экземпляров для обеспечения устойчивой и надежной работы сервисов.

Блок управления контейнерами тесно взаимодействует с блоком интерфейса и взаимодействия пользователя, принимая от него команды и передавая обратную связь об успешности выполнения операций. Это позволяет пользователям не только запускать или останавливать контейнеры, но и отслеживать их состояние через пользовательский интерфейс.

Связь с блоком операций масштабирования имеет решающее значение для поддержания эффективности работы приложений. Блок управления контейнерами получает сигналы о необходимости масштабирования от этого модуля, основываясь на анализе текущей нагрузки и производительности контейнеров. При получении указания на масштабирование вверх блок управления контейнерами инициирует создание дополнительных контейнеров, а при масштабировании вниз – останавливает и удаляет избыточные экземпляры. Это динамическое взаимодействие позволяет системе быстро адаптироваться к изменяющимся требованиям нагрузки, поддерживая оптимальную производительность и уровень ресурсов.

Также блок управления контейнерами напрямую взаимодействует с блоком контейнеров, осуществляя непосредственное управление их жизненным циклом.

Блок управления контейнерами представляет собой центральное звено между оперативными потребностями приложений и динамическими возможностями инфраструктуры, позволяя эффективно адаптироваться к меняющимся условиям.

2.4 Блок балансировщик нагрузки

Блок балансировщика нагрузки представляет собой системный компонент, выполняющий распределение входящего трафика между запущенными контейнерами, что обеспечивает равномерное распределение нагрузки и предотвращает перегрузку отдельных узлов. Задача этого блока заключается в динамическом анализе текущего состояния нагрузки на контейнеры и их метрик, чтобы решить, какой контейнер наиболее подходит для обработки нового запроса, исходя из стремления к оптимизации производительности на уровне всей системы. В случае сбоя или простоя какого-либо контейнера, балансировщик нагрузки незамедлительно перераспределяет трафик к другим доступным экземплярам, тем самым уменьшая время простоя и поддерживая непрерывность сервиса.

Блок балансировщика нагрузки представляет собой сложный алгоритмический компонент, который интегрируется с блоком контейнеров, непрерывно собирая данные о состоянии и производительности. Это достигается путём циклического обращения к каждому контейнеру с целью получения актуальных метрик, на основании которых и происходит распределение трафика. Важность этого блока подчеркивается его способностью к принятию мгновенных решений об изменении нагрузки, основываясь на реальном времени отклика и текущей доступности контейнеров.

Балансировщик нагрузки постоянно взаимодействует с блоком контейнеров. Связь между балансировщиком нагрузки и контейнерами является основной для его функционирования. Балансировщик постоянно просматривает состояние и нагрузку каждого контейнера, чтобы распределять входящие запросы максимально эффективно. В случае если определённый контейнер становится недоступен или перегружен, балансировщик нагрузки перенаправляет трафик на другие, менее загруженные контейнеры. Это обеспечивает балансировку нагрузки и повышает отказоустойчивость системы.

Блок анализа нагрузки предоставляет данные о текущей загрузке каждого контейнера, и на основании этой информации балансировщик принимает решение о том, какому контейнеру следует обработать новый запрос. Такая взаимосвязь создает основу для эффективного управления ресурсами и гарантирует надежную и сбалансированную работу сервисов.

В целом, балансировщик нагрузки играет ключевую роль в обеспечении высокой производительности, доступности и отказоустойчивости контейнеризированных приложений, являясь неотъемлемой частью современных распределенных систем.

2.5 Блок контейнеров

Контейнеры – это технология, которая позволяет упаковать

приложение и все его зависимости в один компактный исполняемый пакет. Это обеспечивает непрерывность работы приложения при переходе от одной вычислительной среды к другой. Суть контейнеризации заключается в возможности легко и быстро запускать приложения в любом окружении, что особенно ценится в разработке программного обеспечения, тестировании и эксплуатации.

Блок контейнеров является непосредственно исполняемым слоем в архитектуре системы, обрабатывающим пользовательские и системные запросы. Связь с блоком балансировщика нагрузки критически важна, так как блок балансировщика непрерывно распределяет входящие запросы на основе текущей загрузки контейнеров, обеспечивая эффективное и справедливое использование ресурсов и минимизацию времени отклика. Эта динамическая связь позволяет системе поддерживать высокую доступность и производительность, распределяя нагрузку между контейнерами и перенаправляя трафик к менее загруженным экземплярам.

Блок управления контейнерами осуществляет взаимодействие с блоком контейнеров, отправляя команды для создания, запуска, остановки и удаления контейнеров. Это включает в себя обработку запросов на выделение ресурсов и управление жизненным циклом контейнера в ответ на операции пользователей и системных событий. Блок управления является административным центром, через который проходят все операционные команды, связанные с контейнерами.

Блок операций масштабирования взаимодействует с блоком контейнеров, регулируя их количество в соответствии с текущей нагрузкой и потребностями системы. Этот блок принимает решения о масштабировании вверх или вниз, добавляя или удаляя контейнеры, чтобы соответствовать требуемому уровню обслуживания и управления ресурсами. Процесс масштабирования является результатом анализа метрик загрузки и производительности, полученных от блока метрик и блока анализа нагрузки.

Блок метрик тесно связан с блоком контейнеров, поскольку он отвечает за сбор и анализ данных о состоянии и производительности каждого контейнера. Эти данные необходимы для оценки нагрузки на систему и принятия обоснованных решений о распределении ресурсов и масштабировании. Взаимодействие с блоком метрик позволяет блоку контейнеров поддерживать оптимальное состояние работы, предоставляя важную информацию для всех уровней управления системой.

Эти взаимодействия образуют основу для эффективного и гибкого управления приложениями в динамичной, распределенной среде. Контейнеры гарантируют, что приложения могут быть надежно и масштабируемо запущены в любой среде, от локальных рабочих станций до облачных платформ, что является критически важным для современных бизнес-процессов.

2.6 Блок операций масштабирования

Блок операций масштабирования отвечает за динамическое изменение количества контейнеров в системе в зависимости от текущей нагрузки и требований пользователей. Это включает в себя автоматическое увеличение числа контейнеров (масштабирование вверх) при росте нагрузки для поддержания производительности и отказоустойчивости, а также уменьшение количества контейнеров (масштабирование вниз) в периоды низкой активности для экономии ресурсов. Функциональность этого блока критически важна для обеспечения гибкости и эффективности облачных ресурсов, позволяя системе быстро адаптироваться к изменяющимся условиям.

Связи блока операций масштабирования характеризуются активным обменом информацией с другими модулями системы. Он принимает информацию о состоянии контейнеров от блока контейнеров, чтобы определить, какие контейнеры следует запустить или остановить. С блоком управления контейнерами существует двусторонняя связь, блок масштабирования инициирует создание или удаление контейнеров и получает обратную связь о статусе этих операций.

Блок мониторинга предоставляет необходимые данные о производительности и здоровье системы, что позволяет определить необходимость в масштабировании. Взаимодействие с блоком интерфейса и взаимодействия пользователя позволяет администраторам и пользователям инициировать процессы масштабирования вручную на основе их управленческих решений или автоматически через пользовательские настройки.

Кроме того, тесная связь с блоком анализа нагрузки критична для принятия решений о масштабировании, так как этот блок предоставляет аналитические данные о текущей нагрузке, производительности и предсказании тенденций, что позволяет блоку операций масштабирования реагировать оперативно и адекватно.

Эти многосторонние связи создают комплексную систему управления, способную к самонастройке и оптимизации в зависимости от внутренних и внешних изменений в использовании ресурсов.

2.7 Блок анализа нагрузки

Блок анализа нагрузки является ключевым компонентом системы, задачей которого является обработка и интерпретация данных о производительности контейнеров, получаемых от блока метрик. Он использует эти данные для выявления информации, пиковой активности и потенциальных узких мест в инфраструктуре. На основе анализа производительности блок анализа нагрузки предоставляет ценную информацию для принятия обоснованных решений о масштабировании

инфраструктуры, управлении ресурсами и балансировке нагрузки.

Связи блока анализа нагрузки интегрируют его в общую систему управления контейнерами. Он тесно связан с блоком метрик, откуда поступают данные для анализа. Это двусторонняя связь, блок анализа нагрузки не только принимает данные, но и может инициировать сбор дополнительной информации, необходимой для более глубокого анализа. Взаимодействие с блоком операций масштабирования является двунаправленным, блок анализа нагрузки поставляет данные, необходимые для определения моментов масштабирования системы, и, в свою очередь, получает обратную связь о влиянии этих операций на производительность системы.

Помимо этого, блок анализа нагрузки напрямую влияет на работу блока балансировщика нагрузки, предоставляя ему информацию о текущей загрузке каждого контейнера и помогая эффективно распределять входящий трафик между контейнерами. Это позволяет балансировщику нагрузки оптимизировать распределение запросов в реальном времени, повышая производительность и снижая задержки.

Таким образом, блок анализа нагрузки выполняет функцию интеллектуального анализа, который необходим для прогнозирования и планирования ресурсов системы, делая процесс масштабирования и балансировки более предсказуемым и эффективным. Эти аналитические способности обеспечивают решения, которые помогают системе быть адаптивной к изменяющимся требованиям и обеспечивать высокий уровень обслуживания пользователей.

2.8 Блок метрик

Блок метрик – это инструментарий системы управления контейнерами, который отвечает за сбор, агрегацию и предоставление данных о производительности и ресурсах контейнеров. Он играет роль центрального репозитория, где хранится вся информация о CPU, памяти, хранилище, сетевой активности и других критически важных метриках контейнеров. Блок метрик активно собирает данные в реальном времени, обеспечивая актуальность информации о состоянии системы для других модулей.

Связь блока метрик с блоком анализа нагрузки является однонаправленной: он поставляет данные, необходимые для анализа текущей производительности и прогнозирования будущих трендов нагрузки, что позволяет принимать информированные решения о масштабировании и распределении ресурсов. Эти данные также используются для оптимизации работы блока балансировщика нагрузки, позволяя ему эффективно распределять запросы на основе актуальной нагрузки на каждый контейнер.

Блок метрик также связан с блоком контейнеров, откуда он напрямую получает информацию о состоянии и работе каждого контейнера. Эта двусторонняя связь позволяет блоку метрик активно отслеживать состояние

каждого контейнера, обновлять метрики и посылать уведомления о важных событиях или изменениях в состоянии контейнеров.

Кроме того, блок метрик передаёт данные в блок мониторинга, где эти данные могут быть использованы для визуализации состояния системы, уведомления пользователей о критических событиях и долгосрочного анализа производительности для планирования масштабирования инфраструктуры.

В целом, блок метрик является жизненно важным для оперативного реагирования на изменения в работе системы, постоянно предоставляя данные, которые нужны для поддержания высокой производительности, надёжности и доступности сервисов, предоставляемых контейнеризированными приложениями.

2.9 Блок мониторинга

Блок мониторинга в системе управления контейнерами отвечает за наблюдение, отслеживание состояния и предоставление своевременной информации о здоровье и производительности системы. Этот компонент является жизненно важным для оперативного управления и поддержания надежности сервисов, поскольку он выявляет потенциальные проблемы и неэффективное использование ресурсов до того, как они приведут к сбоям или ухудшению качества обслуживания. Блок мониторинга использует различные инструменты и технологии для сбора логов, проверки работоспособности контейнеров и их компонентов, а также оценки нагрузки и производительности системы.

Связь блока мониторинга с блоком метрик является входящей: он принимает собранные метрики и использует их для генерации уведомлений, алертов и отчётов, которые могут быть представлены администраторам системы через пользовательский интерфейс или другие каналы оповещения. Эти данные помогают администраторам оценить текущее состояние системы и принимать решения об устранении неполадок или оптимизации ресурсов.

В свою очередь, блок мониторинга взаимодействует с блоком контейнеров, направляя данные о работоспособности и доступности каждого контейнера в блок управления контейнерами. Это позволяет своевременно реагировать на любые сбои или проблемы в работе контейнеров, например, инициировать их перезапуск или масштабирование.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Общая функциональная структура проекта аналогична структурной, но она не включает в себя компоненты представлений из-за декларативного стиля написания. Взаимосвязь между основными компонентами представлена на диаграмме классов ГУИР.400201.071 РР.1.

В контексте данного проекта используется концепция протоколов, которая играет ключевую роль в организации архитектуры и обеспечении гибкости разработки. Протоколо-ориентированная парадигма предлагает подход, сосредотачивающий внимание на определении интерфейсов, которые объекты должны реализовывать, вместо того чтобы ориентироваться на конкретные типы данных или иерархию наследования, характерную для классического объектно-ориентированного программирования.

Хотя язык Python, применяемый в данном проекте, не включает в себя встроенную поддержку протоколов на уровне языка, как это реализовано, например, в Swift, было принято решение о применении подходов, которые эмулируют концепцию протоколов. Для этого используются инструменты, такие как абстрактные базовые классы (ABC) и декораторы.

Основная идея заключается в том, чтобы через использование этих механизмов определить интерфейсы, которые должны быть реализованы классами или структурами данных, независимо от их типа или наследования. Это позволяет разработчикам создавать абстракции, которые зависят только от интерфейса, а не от конкретной реализации, что делает код более гибким и поддающимся расширению.

Применение протоколов в разработке программного обеспечения имеет существенные преимущества, которые оказывают положительное влияние на процесс разработки, читаемость кода и его обслуживание.

Во-первых, разделение интерфейса и его реализации способствует повышению читаемости и структурированности кода. Интерфейсы определяют контракты, которые должны быть выполнены классами или структурами данных, а реализация интерфейсов находится в отдельных компонентах. Это делает код более модульным, понятным и легко поддерживаемым, так как разработчики могут легко найти и изменить необходимые части без необходимости вникать в детали реализации.

Во-вторых, использование протоколов способствует созданию более модульной архитектуры приложения, что особенно важно в крупных проектах. Благодаря высокой степени абстракции и расширяемости, протоколы позволяют минимизировать влияние изменений в одной части кода на другие его компоненты. Это делает проект более устойчивым к изменениям и облегчает поддержку в процессе его развития.

Кроме того, применение протоколов существенно ускоряет процесс разработки программ и упрощает тестирование кода. Благодаря четкому определению интерфейсов и их независимости от конкретной реализации, разработчики могут параллельно работать над разными частями системы, что

повышает производительность и снижает вероятность ошибок.

Таким образом, можно выделить следующую функциональную структуру проекта: сервисные классы и дополнительные классы.

К сервисным классам относятся:

– containerService;
– metricsService;
– scaleService;
– loadBalancer;

К классам данных и моделей относятся:

– container;
– containerCreate;
– containerBase;
– scale.

3.1 Сервисные классы

3.1.1 ContainerService

ContainerService является ключевым модулем в архитектуре приложения, ответственным за управление контейнерами. Он обеспечивает интерфейс для создания, запуска, остановки и удаления контейнеров, что позволяет приложению динамически масштабироваться в зависимости от текущей нагрузки. Сервис также отслеживает активные контейнеры и обновляет этот список при необходимости, обеспечивая актуальность информации о состоянии контейнеров.

Одной из важных функций ContainerService является обработка ошибок и исключений, которые могут возникнуть в процессе работы с контейнерами. Сервис предоставляет удобный интерфейс для обработки таких ситуаций и уведомления об ошибках, что повышает надежность и устойчивость приложения. Кроме того, ContainerService принимает меры для обеспечения безопасности контейнеров и окружения, включая настройку доступа и управление привилегиями.

В целом, ContainerService играет важную роль в обеспечении надежной работы контейнеризованных сервисов и облегчает процесс развертывания и масштабирования приложения. Его функциональность и надежность являются ключевыми аспектами. ContainerService содержит атрибуты:

– `containers;`
– `initial_containers_count;`
– `container_cycle.`

Атрибут `containers` является основой для управления контейнерами в рамках сервиса. Он представляет собой список объектов контейнеров, где каждый элемент списка является экземпляром класса Container. Этот список обеспечивает сервису доступ ко всем контейнерам, которыми он управляет, и содержит подробную информацию о каждом контейнере.

Каждый объект контейнера содержит в себе различные атрибуты, которые описывают состояние и характеристики контейнера. Некоторые из этих атрибутов включают идентификатор контейнера, его текущий статус (например, запущен или остановлен), URL, а также другие метаданные, необходимые для управления контейнером.

Список `containers` обеспечивает эффективное управление всеми контейнерами в рамках сервиса. Он позволяет сервису выполнять различные операции над контейнерами, такие как создание, удаление, запуск, остановка и мониторинг. Кроме того, этот список может быть использован для выполнения различных запросов и операций, связанных с контейнерами, что делает его ключевым компонентом в функциональной структуре сервиса.

Атрибут `initial_containers_count` является важной характеристикой сервиса, определяющей начальное количество контейнеров, которые будут созданы и управляются сервисом при его инициализации. Это число задается в момент запуска сервиса и используется для установки начальной базовой точки для управления контейнерами и масштабирования их количества.

Инициализация сервиса с определенным количеством контейнеров позволяет ему быть готовым к обработке определенного объема запросов и нагрузки уже на старте. Кроме того, установка начального количества контейнеров позволяет более эффективно управлять их созданием и развертыванием, так как это число является отправной точкой для дальнейшего масштабирования.

Атрибут `initial_containers_count` также играет важную роль в мониторинге и управлении изменениями в количестве контейнеров во времени. Путем отслеживания начального числа контейнеров можно определять тренды и паттерны изменений в нагрузке на сервис, что помогает принимать решения о масштабировании и оптимизации ресурсов.

Таким образом, атрибут `initial_containers_count` обеспечивает основу для эффективного управления и масштабирования контейнеров в рамках сервиса, позволяя ему адаптироваться к изменяющейся нагрузке и обеспечивать стабильную и надежную работу.

`Container_cycle` представляет собой механизм, который обеспечивает эффективное управление и повторное использование контейнеров в рамках сервиса. Этот атрибут представляет собой итерируемый объект, который предоставляет доступ к контейнерам в списке в циклическом порядке.

Основная идея `Container_cycle` заключается в том, чтобы создать циклический итератор по списку контейнеров, позволяя выполнять действия для каждого контейнера в определенном порядке или в цикле. Например, если в рамках сервиса требуется выполнить какое-то действие для каждого контейнера, например, проверить их состояние или выполнить обслуживание, можно использовать `Container_cycle` для обхода контейнеров в цикле.

При достижении конца списка контейнеров итератор начинает снова с

начала списка, что позволяет повторно использовать контейнеры без необходимости создавать новые. Это особенно полезно в ситуациях, когда контейнеры могут быть запущены, остановлены и запущены снова, и нужно обеспечить равномерное использование ресурсов и балансировку нагрузки между ними.

Таким образом, `Container_cycle` обеспечивает эффективное управление контейнерами в рамках сервиса, обеспечивая повторное использование и равномерное распределение нагрузки, что способствует оптимальной работе системы.ContainerService содержит методы:

- `update_containers_list();`
- `list_all_containers();`
- `create_container();`
- `list_active_containers ();`
- `delete_container ();`
- `start_container ();`
- `stop_container ();`
- `get_container_logs ();`
- `get_containers_count ();`
- `get_containers_count_by_image ();`
- `get_containers_by_image();`
- `restart_failed_containers ()`.

Метод `update_containers_list()` играет важную роль в поддержании актуальности данных о контейнерах в сервисе. Его целью является обновление списка контейнеров, используемого сервисом, чтобы он отражал текущее состояние контейнеров.

При вызове этого метода сервис запрашивает у Docker информацию о всех контейнерах, находящихся на хосте, включая как запущенные, так и остановленные. Затем полученная информация используется для обновления внутреннего списка контейнеров в сервисе.

Этот процесс обновления особенно важен, поскольку состояние контейнеров может меняться во времени: они могут быть созданы, запущены, остановлены или удалены другими процессами или пользователями. Поэтому для обеспечения корректной работы сервиса необходимо регулярно обновлять информацию о контейнерах.

После выполнения метода `update_containers_list()` список контейнеров в сервисе будет содержать последние данные о контейнерах, что позволит другим частям приложения использовать актуальную информацию о контейнерах для принятия решений и выполнения операций, таких как создание, запуск, остановка или удаление контейнеров.

Метод `list_all_containers()` представляет собой важное средство для получения обзора текущего состояния всех контейнеров на хосте. Он осуществляет запрос к Docker API для извлечения информации обо всех контейнерах, находящихся на данном хосте. Возвращаемый результат

представляет собой список объектов типа Container, где каждый объект содержит информацию об отдельном контейнере.

При вызове метода `list_all_containers()` Docker API возвращает информацию обо всех контейнерах, включая как активные (запущенные), так и неактивные (остановленные) контейнеры. Для каждого контейнера извлекается и сохраняется следующая информация:

– идентификатор контейнера `id`: уникальный идентификатор, который однозначно идентифицирует контейнер;

– образ контейнера `image`: название образа, на основе которого был создан данный контейнер;

– статус контейнера `status`: текущий статус контейнера, такой как «running» (запущен) или «exited» (остановлен);

– url контейнера `url`: url-адрес, по которому можно обратиться к контейнеру для выполнения операций или получения дополнительной информации;

– метаданные контейнера `labels`: дополнительные метаданные или метки, присвоенные контейнеру при его создании или запуске.

В результате выполнения метода `list_all_containers()` возвращается список объектов типа Container, где каждый объект представляет один контейнер и содержит вышеуказанную информацию о нем. Этот список может быть использован другими частями приложения для дальнейшего анализа, мониторинга или управления контейнерами на хосте Docker.

Метод `create_container(container_data: ContainerCreate)` играет ключевую роль в создании нового контейнера на основе предоставленных данных, содержащихся в объекте ContainerCreate. Этот метод принимает экземпляр ContainerCreate, который содержит информацию о необходимых параметрах для создания контейнера, таких как образ, команда, метки и другие настройки.

При вызове этого метода происходит инициирование процесса создания контейнера на хосте Docker. Для этого метод обращается к Docker API и передает ему необходимые данные для создания нового контейнера в соответствии с указанными параметрами. Docker API обрабатывает запрос и выполняет создание контейнера на основе предоставленных данных.

В случае успешного создания контейнера, метод возвращает объект типа Promise, который представляет асинхронную операцию. Этот Promise разрешится в объект типа Container, который содержит информацию о только что созданном контейнере.

В процессе создания контейнера могут возникнуть различные ошибки, такие как отсутствие указанного образа, недостаточность ресурсов на хосте Docker или проблемы с сетевым взаимодействием. В случае возникновения таких ошибок, метод может сгенерировать исключение или вернуть объект Promise с состоянием "rejected", содержащий информацию об ошибке.

Использование асинхронного подхода через Promise позволяет эффективно управлять процессом создания контейнера и обрабатывать результаты операции, даже если она занимает некоторое время. Это позволяет программе эффективно работать с созданием контейнеров и обеспечивает гибкость в управлении контейнеризированными приложениями.

Метод `list_active_containers()` является важным инструментом для получения списка активных контейнеров на хосте Docker. В отличие от метода `list_all_containers()`, который возвращает информацию обо всех контейнерах, включая и те, которые были остановлены, `list_active_containers()` извлекает только данные о запущенных контейнерах.

При вызове этого метода происходит обращение к Docker API для получения списка контейнеров, которые в данный момент находятся в активном состоянии. То есть, это контейнеры, которые были запущены и находятся в процессе выполнения.

Возвращаемый результат метода представляет собой список объектов типа Container, где каждый объект содержит информацию о конкретном активном контейнере. Эти объекты предоставляют доступ к различным атрибутам контейнера, таким как его идентификатор, используемый образ, статус и другие важные параметры.

Использование метода `list_active_containers()` позволяет получить только актуальную информацию о работающих контейнерах, что может быть полезным при управлении контейнеризированными приложениями. Например, это может быть полезно для мониторинга состояния запущенных контейнеров, выполнения операций на них или принятия решений на основе их текущего состояния.

Метод `delete_container(container_id: string)` выполняет важную функцию удаления контейнера с указанным идентификатором. Как только этот метод вызывается с идентификатором контейнера в качестве аргумента, он обращается к Docker API для поиска контейнера с указанным идентификатором и последующего его удаления.

При вызове этого метода происходят различные процессы: метод получает идентификатор контейнера в качестве аргумента. Затем он использует этот идентификатор для обращения к Docker API с запросом на удаление контейнера. Docker API находит контейнер по предоставленному идентификатору и удаляет его с хоста Docker.

Важно отметить, что удаление контейнера обычно включает остановку контейнера, если он запущен, а затем удаление его с хоста. Если контейнер не найден с указанным идентификатором или возникают проблемы при удалении, метод может сгенерировать исключение или вывести сообщение об ошибке, в зависимости от его реализации.

Использование метода `delete_container()` позволяет эффективно управлять контейнерами, освобождая ресурсы хоста Docker и поддерживая

порядок в системе. Это может быть полезно, когда контейнеры становятся ненужными или устаревшими и требуют удаления для освобождения места или ресурсов для других задач.

Метод `start_container(container_id: string)` играет ключевую роль в управлении жизненным циклом контейнеров в среде Docker. Его целью является запуск конкретного контейнера, идентификатор которого передается в качестве аргумента. Когда вызывается этот метод, система начинает процесс запуска указанного контейнера, что включает в себя инициализацию необходимых ресурсов, настройку сетевых соединений и запуск приложения, определенного внутри контейнера.

Для начала метод получает идентификатор контейнера в виде строки. Затем, используя этот идентификатор, он обращается к Docker API для поиска соответствующего контейнера. После нахождения контейнера система инициирует процесс его запуска, который включает в себя загрузку необходимого образа, настройку окружения и запуск заданных команд или приложений внутри контейнера.

Ключевым моментом является обработка возможных ошибок в процессе запуска контейнера. Если образ контейнера не найден или возникают проблемы с его конфигурацией, метод может сгенерировать исключение или вернуть соответствующее сообщение об ошибке. В таких случаях важно предусмотреть механизм обработки ошибок для корректной обработки и восстановления работы системы.

Использование метода `start_container()` дает возможность динамического управления контейнерами, позволяя легко масштабировать и обновлять инфраструктуру приложения по мере необходимости. Он является неотъемлемой частью процесса автоматизации развертывания и обеспечивает гибкость и эффективность в управлении контейнеризированными приложениями.

Метод `stop_container(container_id: string)` представляет собой важное средство управления жизненным циклом контейнеров в Docker-среде. Его основная задача заключается в остановке работы контейнера, идентификатор которого передается в качестве параметра метода.

При вызове этого метода система обращается к Docker API для поиска контейнера с указанным идентификатором. После нахождения контейнера происходит его остановка, которая включает в себя завершение всех работающих процессов внутри контейнера и освобождение используемых ресурсов.

Важно отметить, что метод `stop_container()` играет ключевую роль в обеспечении контроля над ресурсами и безопасностью в Docker-среде. Он позволяет предотвратить нецелесообразное использование ресурсов системы и управлять состоянием контейнеров в соответствии с требованиями приложения или системы.

Обработка возможных ошибок также является важной частью этого

метода. Если возникают проблемы при остановке контейнера, такие как отсутствие доступа к Docker API или ошибки во время выполнения операции, необходимо предусмотреть соответствующие механизмы обработки ошибок для обеспечения корректной работы системы.

Использование метода `stop_container()` предоставляет возможность эффективного управления ресурсами и обеспечивает безопасность и надежность работы контейнеризированных приложений в среде Docker.

Метод `get_container_logs(container_id: string)` предоставляет возможность получения журналов работы контейнера с указанным идентификатором. При вызове этого метода система обращается к Docker API для извлечения журналов указанного контейнера.

После успешного запроса к API и получения журналов метод возвращает объект `Promise`. Этот объект представляет собой асинхронную операцию, которая при завершении разрешится в объект ContainerLog. Объект ContainerLog содержит информацию о журналах контейнера, которые могут быть использованы для анализа работы приложения в контейнере, выявления ошибок или отладки.

Важно отметить, что асинхронная природа метода `get_container_logs()` позволяет получать журналы контейнера без блокировки основного потока выполнения программы. Это особенно полезно в случаях, когда необходимо получить журналы из большого числа контейнеров или при работе с другими асинхронными операциями.

Обработка возможных ошибок также важна для корректной работы метода. Если возникают проблемы при получении журналов контейнера, например, из-за недоступности Docker API или ошибок во время выполнения операции, необходимо предусмотреть соответствующие механизмы обработки ошибок для обеспечения надежности работы системы.

Использование метода `get_container_logs()` помогает в мониторинге работы контейнеризованных приложений, анализе и отладке их работы, а также обеспечивает более эффективное управление и отслеживание состояния контейнеров в Docker-среде.

Метод `get_containers_count()` предоставляет информацию о общем количестве контейнеров, находящихся на хосте Docker. При вызове этого метода система обращается к Docker API для получения списка всех контейнеров и подсчета их общего числа.

После получения списка контейнеров метод подсчитывает количество элементов в этом списке и возвращает это число в качестве результата. Таким образом, клиентский код может получить общее количество контейнеров на хосте Docker, используя данный метод.

Важно отметить, что метод `get_containers_count()` может использоваться для мониторинга загруженности хоста Docker и управления им. Например, на основе этой информации можно принимать решения о масштабировании приложений, оптимизации ресурсов или управлении

жизненным циклом контейнеров.

Использование метода `get_containers_count()` позволяет эффективно управлять контейнерами в Docker-среде, контролировать их количество и обеспечивать необходимую производительность и надежность системы.

Метод `get_containers_count_by_image(image_name: string)` предоставляет информацию о количестве контейнеров на хосте Docker, созданных с использованием определенного образа. При вызове этого метода система обращается к Docker API для получения списка всех контейнеров, затем фильтрует этот список, чтобы оставить только те контейнеры, которые созданы с указанным образом.

После фильтрации списка контейнеров метод подсчитывает количество элементов в этом списке и возвращает это число в качестве результата. Таким образом, клиентский код может узнать количество контейнеров, созданных с определенным образом, используя данный метод.

Этот метод полезен для мониторинга использования конкретных образов на хосте Docker. Например, он может использоваться для определения, сколько экземпляров приложения запущено с определенной версией образа, или для выявления неиспользуемых образов, которые можно удалить для освобождения ресурсов.

Использование метода `get_containers_count_by_image(image_name: string)` помогает эффективно управлять контейнерами и ресурсами в Docker-среде, обеспечивая более точное мониторинг и управление использованием образов.

Метод `get_containers_by_image(image_name: string)` предоставляет возможность получить список контейнеров, которые были созданы с использованием определенного образа Docker. При вызове этого метода система осуществляет запрос к Docker API для получения списка всех контейнеров, а затем фильтрует этот список, оставляя только те контейнеры, которые были созданы с указанным образом.

После фильтрации списка контейнеров метод возвращает этот список клиентскому коду. Это позволяет приложениям и сервисам получить доступ к информации о контейнерах, связанных с определенным образом, и произвести необходимые операции с этими контейнерами.

Использование метода `get_containers_by_image(image_name: string)` может быть полезным для мониторинга и управления контейнерами, созданными с определенным образом. Например, этот метод может использоваться для проверки статуса и работы экземпляров приложения, созданных с определенной версией образа, или для выполнения операций управления контейнерами, таких как запуск, остановка или удаление.

Таким образом, метод `get_containers_by_image(image_name: string)` представляет собой важный инструмент для управления контейнерами и ресурсами в среде Docker, обеспечивая возможность

эффективного мониторинга и управления контейнерами, связанными с определенным образом Docker.

Метод `restart_failed_containers()` представляет собой процесс автоматического перезапуска контейнеров, которые завершились с ошибкой. Он циклически проверяет состояние каждого контейнера на хосте Docker и, если обнаруживает, что контейнер завершил свою работу с ошибкой, то запускает процедуру его перезапуска.

При каждой итерации цикла метод извлекает текущее состояние каждого контейнера, анализирует его статус и код завершения. Если контейнер находится в состоянии "exited" и имеет ненулевой код завершения, что указывает на возникновение ошибки, метод инициирует перезапуск контейнера.

В процессе перезапуска метод вызывает соответствующие Docker API для запуска контейнера заново, восстанавливая его работоспособность. После успешного перезапуска контейнера метод продолжает цикл, переходя к следующему контейнеру.

Метод `` `restart_failed_containers()` `` обеспечивает автоматическое восстановление работоспособности контейнеров после их аварийного завершения, что является важным механизмом для обеспечения непрерывности работы приложений и сервисов, запущенных в контейнерах Docker. Этот процесс позволяет поддерживать стабильность и доступность приложений, минимизируя влияние возможных сбоев или ошибок в работе контейнеров.

3.1.2 MetricsService

MetricsService является ключевым компонентом системы, ответственным за сбор, обработку и анализ метрик контейнеров. Его функциональность нацелена на обеспечение контроля за производительностью и состоянием контейнеров, а также на выявление потенциальных проблем или узких мест в их работе.

Основной задачей MetricsService является сбор статистических данных о ресурсах, используемых контейнерами. Эти данные включают информацию о загрузке CPU, использовании памяти, сетевом трафике и других параметрах, влияющих на работу контейнеров. Для сбора этих данных MetricsService взаимодействует с Docker API, получая актуальную информацию о состоянии контейнеров.

После сбора метрик MetricsService осуществляет их анализ. Это включает в себя вычисление средних значений, максимальных и минимальных значений, а также определение трендов и аномалий. Например, MetricsService может выявлять периоды повышенной нагрузки на CPU или неожиданные колебания использования памяти, что может свидетельствовать о проблемах в работе контейнеров.

Результаты анализа метрик используются для принятия решений о дальнейших действиях. Например, если MetricsService обнаруживает перегрузку какого-то контейнера, он может автоматически масштабировать его, чтобы справиться с повышенной нагрузкой. Также метрики могут быть использованы для оптимизации распределения нагрузки между контейнерами или для выявления узких мест в инфраструктуре.

Таким образом, MetricsService играет важную роль в обеспечении стабильной и эффективной работы контейнеров, обеспечивая контроль за их производительностью и ресурсами. Он является неотъемлемой частью системы управления контейнерами, обеспечивая оперативное реагирование на изменяющиеся условия и потребности приложений. Основные методы класса MetricsService являются:

– `get_container_stats(container_id: str;`
– `analyze_stats(stats: Dict[str, Any]).`

Метод `get_container_stats(container_id: str)` класса MetricsService предназначен для получения статистики о ресурсах для конкретного контейнера по его идентификатору. Этот метод принимает идентификатор контейнера в качестве аргумента и использует Docker API для запроса статистических данных о данном контейнере.

После получения идентификатора контейнера, метод формирует запрос к Docker API для извлечения информации о его ресурсах, таких как использование CPU, памяти, сети и др. Затем API Docker возвращает данные о статистике контейнера в виде словаря, содержащего различные метрики и их значения.

Полученные данные о статистике обычно включают в себя информацию о загрузке CPU, использовании памяти, сетевом трафике и других ресурсах, в зависимости от настроек контейнера и конфигурации Docker.

После получения статистики о контейнере, эти данные могут быть переданы для дальнейшего анализа методу `analyze_stats`.

Использование метода `get_container_stats()` позволяет оперативно получать информацию о текущем состоянии контейнера, что полезно для мониторинга и управления его ресурсами в реальном времени. Этот метод играет важную роль в контроле и оптимизации работы контейнеров, обеспечивая оперативное реагирование на изменения и управление ресурсами в соответствии с требованиями приложения или системы.

Метод `analyze_stats(stats: Dict[str, Any])` в классе MetricsService является ключевым инструментом для интерпретации статистических данных, полученных о ресурсах контейнера из Docker API. Этот метод производит анализ различных метрик, включая использование центрального процессора (CPU), памяти, сети и дискового ввода/вывода (I/O). Принимая на вход словарь `stats`, содержащий статистические данные о контейнере, метод `analyze_stats()` извлекает необходимые значения и

производит вычисления для получения полезной информации о текущем состоянии контейнера.

Во время анализа, метод может вычислять различные показатели, такие как общее использование CPU и его процентное отношение к общей доступной мощности, использование памяти и сети, а также объем дискового ввода/вывода. Эти вычисления позволяют получить представление о нагрузке на контейнер и его ресурсах, что важно для мониторинга и оптимизации работы приложения.

Кроме того, метод `analyze_stats()` может форматировать полученные результаты для удобства восприятия, например, представляя значения в удобных единицах измерения или округляя их до определенного числа знаков после запятой. Это позволяет представить информацию в более удобном виде для дальнейшего анализа или отображения.

В итоге, метод `analyze_stats()` предоставляет важную информацию о состоянии контейнера, которая может быть использована для принятия решений о масштабировании, оптимизации ресурсов и обеспечении надлежащего функционирования приложений в контейнерной среде.

3.1.3 ScaleService

Класс ScaleService является фундаментальным компонентом системы, предназначенной для автоматизированного управления масштабированием контейнеров в приложении. Его функциональность нацелена на обеспечение гибкости и эффективности в распределении нагрузки и ресурсов в зависимости от текущих условий и требований.

Основная цель ScaleService состоит в том, чтобы обеспечить возможность динамического изменения количества экземпляров контейнеров в соответствии с изменяющейся нагрузкой и другими факторами. Это позволяет приложению адаптироваться к изменениям в окружении, спросе пользователей или другим переменным условиям, обеспечивая оптимальное использование ресурсов и высокую производительность.

Ключевые особенности класса ScaleService включают:

– автоматизация масштабирования: ScaleService предоставляет возможность автоматического масштабирования приложения на основе определенных правил или метрик. Это позволяет реагировать на изменения в нагрузке или ресурсах без необходимости вмешательства оператора.

– динамическое масштабирование: класс позволяет динамически управлять количеством экземпляров контейнеров в зависимости от текущей нагрузки и требований. Это позволяет обеспечить эластичность приложения и эффективное использование ресурсов.

– поддержка горизонтального и вертикального масштабирования: ScaleService может масштабировать приложение как по горизонтали (добавление или удаление экземпляров контейнеров), так и по вертикали

(изменение ресурсов каждого контейнера).

– интеграция с метриками и мониторингом: класс взаимодействует с сервисами сбора метрик и мониторинга, чтобы использовать актуальную информацию о нагрузке и производительности при принятии решений о масштабировании.

– гибкие настройки и правила масштабирования: ScaleService позволяет настраивать различные параметры и правила масштабирования в соответствии с потребностями приложения. Это включает в себя определение пороговых значений, стратегий масштабирования и других параметров.

В целом, ScaleService является ключевым инструментом для обеспечения масштабируемости и надежности приложений, работающих в контейнерной среде. Его возможности по автоматизации и динамическому управлению масштабированием помогают обеспечить высокую доступность и производительность приложения в любых условиях эксплуатации.

Основные методы класса ScaleService:

– `scale_up(request: ContainerCreate)`;
– `scale_down()`;
– `scale_container(container_id: str, scale_target: int)`;
– `start_new_containers(image_name: str, count: int)`;
– `stop_excess_containers(containers: list, count: int)`.

Метод `scale_up(request: ContainerCreate)` является ключевым элементом механизма автоматического масштабирования, предоставляемого классом ScaleService. Его функциональность направлена на динамическое увеличение емкости системы путем создания новых контейнеров в ответ на увеличивающуюся нагрузку или требования бизнеса.

При вызове этого метода передается запрос на создание нового контейнера, содержащий необходимые параметры для создания, такие как образ контейнера, команда, метки и другие настройки. Эти данные используются для создания нового контейнера с помощью Docker API. После успешного создания контейнера метод возвращает результат выполнения операции, который может быть дальше обработан или использован в системе.

Основная цель метода `scale_up()`– обеспечить гибкость и масштабируемость системы, позволяя ей адаптироваться к изменяющимся условиям и требованиям. Создание новых контейнеров позволяет распределить нагрузку на несколько экземпляров приложения, что повышает отказоустойчивость и общую производительность системы.

В контексте микросервисной архитектуры, где приложение состоит из набора независимых сервисов, метод `scale_up()` становится важным инструментом для обеспечения масштабируемости и отзывчивости системы на изменения в объеме запросов или требований к приложению.

Благодаря автоматизированному масштабированию, система способна реагировать на изменения нагрузки в реальном времени, оптимизируя использование ресурсов и обеспечивая стабильную работу приложения даже

в условиях значительного увеличения запросов.

Метод `scale_down()` в классе ScaleService выполняет обратную операцию методу `scale_up()`. Его цель заключается в уменьшении количества запущенных контейнеров в системе, когда объем запросов или требования к приложению уменьшаются. Путем остановки и удаления лишних контейнеров этот метод помогает оптимизировать использование ресурсов и экономить вычислительные мощности.

При вызове метода `scale_down()` система анализирует текущую нагрузку и количество активных контейнеров. Если общая загрузка системы снижается и имеется избыточное количество контейнеров, этот метод принимает решение об уменьшении числа контейнеров. Он останавливает и удаляет лишние контейнеры, освобождая ресурсы и снижая нагрузку на систему.

Такой подход к масштабированию позволяет системе адаптироваться к изменениям в нагрузке и эффективно использовать ресурсы, минимизируя издержки на поддержку и обслуживание. Этот метод особенно полезен в среде с переменным объемом трафика, где возникают периоды пиковой активности и спадов, позволяя максимально эффективно использовать вычислительные ресурсы и снижать расходы на обслуживание системы.

Метод `scale_container(container_id: str, scale_target: int)` в классе ScaleService предназначен для динамического масштабирования количества экземпляров контейнера, основываясь на текущей нагрузке или требованиях к системе. Он принимает идентификатор контейнера `container_id` и целевое количество экземпляров `scale_target`. Этот метод регулирует количество запущенных контейнеров с заданным идентификатором образа до указанного значения `scale_target`.

При вызове метода `scale_container()` система анализирует текущее количество запущенных контейнеров с указанным образом и сравнивает его с целевым значением `scale_target`. Если текущее количество контейнеров меньше целевого значения, метод запускает дополнительные экземпляры контейнера с указанным образом до достижения целевого количества. Если текущее количество контейнеров превышает целевое значение, метод останавливает и удаляет лишние экземпляры контейнера, чтобы число контейнеров соответствовало целевому значению.

Такой механизм динамического масштабирования позволяет системе автоматически адаптироваться к изменениям в нагрузке или требованиях, обеспечивая оптимальное использование ресурсов и эффективное функционирование системы. Этот метод является ключевым элементом для обеспечения гибкости и масштабируемости приложения, позволяя ему поддерживать высокую производительность и доступность в любых условиях эксплуатации.

Метод `start_new_containers(image_name: str, count: int)` в классе ScaleService предназначен для запуска новых экземпляров

контейнеров на основе указанного образа `image_name`. Он принимает имя образа `image_name` и количество новых экземпляров `count`, которые необходимо запустить.

При вызове этого метода система создает и запускает указанное количество новых контейнеров с заданным образом. Это позволяет динамически увеличивать количество экземпляров приложения для обработки повышенной нагрузки или расширения функциональности системы.

Запуск новых контейнеров осуществляется с использованием Docker API, который обеспечивает управление контейнерами и их жизненным циклом. Каждый новый контейнер создается на основе указанного образа и автоматически запускается в изолированной среде.

Этот метод является важной частью механизма динамического масштабирования и автоматической адаптации системы к изменяющимся условиям. Он позволяет системе моментально реагировать на изменения в нагрузке и эффективно использовать имеющиеся ресурсы для обеспечения оптимальной производительности и доступности приложения.

Метод `stop_excess_containers(containers: list, count: int)` в классе ScaleService предназначен для остановки излишних контейнеров. Он принимает список контейнеров `containers` и количество контейнеров `count`, которые необходимо остановить.

При вызове этого метода система проверяет список контейнеров и останавливает указанное количество излишних контейнеров, начиная с самых старых или наименее активных. Это позволяет динамически уменьшать количество экземпляров приложения в соответствии с текущей нагрузкой или запросами на масштабирование.

Остановка контейнеров осуществляется с использованием Docker API, который обеспечивает управление контейнерами и их жизненным циклом. Каждый контейнер из списка `containers` останавливается и удаляется из системы.

Этот метод является важной частью механизма динамического масштабирования и автоматической адаптации системы к изменяющимся условиям. Он позволяет системе оптимально использовать ресурсы и эффективно управлять количеством контейнеров для обеспечения оптимальной производительности и доступности приложения.

3.1.4 LoadBalancer

Класс LoadBalancer играет ключевую роль в обеспечении равномерного распределения нагрузки и эффективной обработки запросов к контейнерам в приложении. Его функциональность охватывает широкий спектр задач, начиная от мониторинга активности контейнеров до динамического управления их нагрузкой и масштабированием. Давайте более подробно

рассмотрим каждый аспект этого класса:

– балансировка нагрузки: основная задача LoadBalancer - равномерное распределение запросов между доступными контейнерами. Это позволяет избежать перегрузки отдельных экземпляров приложения и обеспечить более стабильную производительность системы в целом.

– мониторинг активности контейнеров: LoadBalancer следит за состоянием и активностью каждого контейнера в системе. Это включает в себя отслеживание загрузки ресурсов, статуса доступности и других метрик, необходимых для принятия решений о перенаправлении запросов.

– выбор наименее загруженного контейнера: одной из ключевых функций LoadBalancer является выбор контейнера с наименьшей загрузкой для обработки запроса. Этот процесс основывается на анализе метрик и текущего состояния контейнеров, чтобы выбрать оптимальный экземпляр для обработки запроса.

– перенаправление запросов к выбранному контейнеру: после выбора наиболее подходящего контейнера, LoadBalancer направляет запросы к этому контейнеру для обработки. Это может включать в себя перенаправление HTTP-запросов, TCP-соединений или других типов запросов в зависимости от конфигурации и требований приложения.

– автоматическое масштабирование: LoadBalancer также может играть важную роль в автоматическом масштабировании системы в зависимости от изменений в нагрузке. Он может реагировать на увеличение или уменьшение запросов, а также на изменения состояния контейнеров, и динамически масштабировать количество экземпляров приложения для оптимизации производительности и использования ресурсов.

– конфигурация и настройка: LoadBalancer обычно предоставляет широкие возможности для настройки и конфигурации своего поведения в соответствии с требованиями приложения. Это может включать в себя определение стратегий балансировки, настройку пороговых значений для масштабирования и другие параметры, позволяющие оптимизировать работу системы.

Итак, класс LoadBalancer является важным элементом инфраструктуры приложения, обеспечивающим стабильную и эффективную работу приложения в условиях изменяющейся нагрузки и требований. Его функциональность включает в себя широкий спектр задач, начиная от балансировки нагрузки и мониторинга активности контейнеров до автоматического масштабирования и настройки поведения системы.

Основные методы класса:

- `__init__`;
- `handle_docker_events`;
- `get_least_loaded_container`;
- `proxy_request`;
- `check_and_scale`;

– get_average_cpu_load;
– get_cpu_load.

В методе __init__ происходит инициализация экземпляра класса LoadBalancer. Сначала создается новый поток, который будет выполнять функцию handle_docker_events для мониторинга событий Docker. Этот поток запускается в фоновом режиме (daemon=True), что позволяет ему работать параллельно с основным потоком выполнения программы.

Далее создается асинхронная задача asyncio.create_task для выполнения метода check_and_scale. Этот метод отвечает за периодическую проверку средней загрузки процессора и масштабирование системы в зависимости от этой загрузки.

Использование отдельного потока для мониторинга событий Docker позволяет обновлять список контейнеров в фоновом режиме, не блокируя основной поток выполнения программы. Таким образом, балансировщик нагрузки может непрерывно отслеживать состояние контейнеров и адаптироваться к изменениям в системе.

Асинхронная задача check_and_scale обеспечивает автоматическое масштабирование системы в зависимости от средней загрузки процессора. Это позволяет оптимизировать использование ресурсов и поддерживать стабильную производительность системы, динамически масштабируя количество контейнеров в соответствии с текущей нагрузкой.

Метод handle_docker_events отвечает за обработку событий Docker. Он используется для мониторинга событий Docker, таких как создание, запуск или остановка контейнеров, а также других изменений, происходящих в Docker-окружении.

Этот метод работает асинхронно и выполняется в отдельном потоке. Он ожидает события от Docker, декодирует их и обрабатывает в соответствии с логикой приложения. В данном случае, когда обнаруживается событие связанное с контейнерами (например, создание или удаление), метод обновляет список контейнеров, вызывая метод update_containers_list() у экземпляра ContainerService.

Этот механизм позволяет балансировщику нагрузки быть в курсе изменений, происходящих в Docker-окружении, и соответствующим образом адаптироваться к этим изменениям. Например, если добавляется новый контейнер, он будет включен в список контейнеров, доступных для обработки запросов, или если контейнер удаляется, он будет исключен из списка.

Метод get_least_loaded_container используется для определения контейнера с наименьшей загрузкой процессора в текущем списке контейнеров. Он осуществляет асинхронный обход всех контейнеров, получает метрики производительности для каждого контейнера и сравнивает их, чтобы найти контейнер с наименьшей загрузкой процессора.

Для каждого контейнера метод запрашивает метрики

производительности, такие как использование CPU, используя функцию `get_container_metrics`. Затем он анализирует полученные данные и выбирает контейнер с наименьшей или равной загрузкой процессора. Если несколько контейнеров имеют одинаковую минимальную загрузку, метод выбирает один из них случайным образом.

Этот метод играет ключевую роль в механизме балансировки нагрузки, поскольку он позволяет выбирать контейнер с наименьшей нагрузкой для обработки каждого запроса. Таким образом, он помогает распределять нагрузку равномерно между контейнерами и обеспечивать оптимальное использование ресурсов.

Метод `proxy_request` выполняет проксирование запроса к наименее загруженному контейнеру. После определения контейнера с наименьшей загрузкой метод формирует URL для перенаправления запроса к этому контейнеру.

Затем метод выполняет асинхронный HTTP-запрос к целевому URL с использованием библиотеки httpx. Он ожидает ответ от контейнера и анализирует его. Если ответ имеет тип JSON, метод возвращает данные JSON, включая идентификатор контейнера. В противном случае, если ответ не является JSON, метод возвращает текстовый ответ вместе с идентификатором контейнера.

Этот метод играет важную роль в процессе балансировки нагрузки и маршрутизации запросов. Он обеспечивает перенаправление каждого запроса к наименее загруженному контейнеру, что помогает распределять нагрузку равномерно и обеспечивать оптимальное использование ресурсов.

Метод `check_and_scale` представляет собой циклическую задачу, которая регулярно проверяет среднюю загрузку процессора на всех контейнерах и принимает решение о масштабировании. Внутри этого метода сначала получается средняя загрузка процессора на всех контейнерах с помощью метода `get_average_cpu_load`.

Далее метод проверяет значение средней загрузки и, если оно превышает установленный порог, инициирует масштабирование вверх. Для этого создается новый контейнер с помощью объекта ContainerCreate, который содержит информацию о желаемом образе для масштабирования. Если средняя загрузка ниже другого порога, метод инициирует масштабирование вниз, удаляя лишние контейнеры.

Этот метод обеспечивает автоматическое масштабирование приложения в зависимости от текущей нагрузки, что позволяет оптимизировать использование ресурсов и поддерживать стабильную производительность системы.

Метод `get_average_cpu_load` выполняет расчет средней загрузки процессора на всех активных контейнерах. Сначала он извлекает список идентификаторов активных контейнеров с помощью метода `list_active_containers` из сервиса контейнеров. Затем для каждого

контейнера из списка он вызывает метод `get_cpu_load`, чтобы получить текущую загрузку процессора.

Далее метод вычисляет суммарную загрузку процессора на всех контейнерах и количество активных контейнеров, учитываемых при расчете. После этого метод проверяет, была ли получена информация о загрузке процессора для хотя бы одного контейнера, чтобы избежать деления на ноль.

Если данные о загрузке процессора доступны хотя бы для одного контейнера, метод вычисляет среднюю загрузку процессора на всех контейнерах путем деления суммарной загрузки на количество контейнеров.

Возвращаемое значение представляет собой среднюю загрузку процессора на всех активных контейнерах в процентном выражении.

Метод `get_cpu_load(container_id)` получает текущую загрузку процессора для контейнера с указанным идентификатором `container_id`.

Сначала метод пытается получить объект контейнера по его идентификатору, используя Docker API. Затем он извлекает статистику процессора для контейнера, включая данные о потреблении CPU и системном потреблении CPU.

Далее метод вычисляет изменение потребления CPU с момента предыдущего измерения и общее системное потребление CPU. Затем он вычисляет процент использования CPU путем деления изменения потребления CPU на общее системное потребление CPU и умножения на количество доступных процессоров.

Если какие-либо данные отсутствуют или происходит ошибка в процессе получения статистики, метод выводит сообщение об ошибке и возвращает None.

Возвращаемое значение метода представляет собой процент использования CPU для указанного контейнера.

3.2 Классы данных и моделей

3.2.1 Container

Container является фундаментальным элементом в системе управления контейнерами проекта. Этот класс представляет собой абстракцию контейнера Docker, позволяя программному обеспечению взаимодействовать с контейнерами, управлять их состоянием и получать информацию о них. Он играет ключевую роль в обеспечении надежности, масштабируемости и гибкости при управлении контейнерами в различных сценариях.

Одной из важнейших функций класса Container является возможность управления жизненным циклом контейнера. Это включает в себя создание новых контейнеров на основе заданных параметров, их запуск, остановку и удаление в соответствии с требованиями приложения. Кроме того, класс Container позволяет получать доступ к журналам контейнеров, что особенно

важно для мониторинга и диагностики работы контейнеризированных приложений.

Благодаря функциональности класса Container, система может эффективно управлять контейнерами как в индивидуальном, так и в коллективном аспекте. Это позволяет автоматизировать процессы развертывания, масштабирования и обслуживания приложений в контейнерах, сокращая время и ресурсы, затрачиваемые на управление инфраструктурой. Кроме того, класс Container обеспечивает единый интерфейс для взаимодействия с контейнерами, что упрощает разработку и поддержку приложений в среде контейнеризации.

Класс Container содержит следующие атрибуты:

– `id`;
– `status`;
– `url`.

Атрибут `id` представляет собой уникальный идентификатор контейнера в системе. Этот идентификатор используется для однозначного идентифицирования каждого контейнера на хосте Docker. В контексте взаимодействия с Docker API, `id` является строковым значением, которое назначается каждому созданному контейнеру.

Уникальность идентификатора контейнера обеспечивается Docker Engine при создании контейнера. Он гарантирует, что каждый контейнер на хосте имеет свой собственный уникальный `id`, который не пересекается с идентификаторами других контейнеров. Это позволяет идентифицировать контейнеры в системе и выполнять с ними различные операции, такие как управление и мониторинг их состояния.

При работе с контейнерами через Docker API, идентификатор контейнера используется для выполнения различных действий. Например, при запуске контейнера или запросе его статуса, Docker API принимает идентификатор контейнера в качестве аргумента, чтобы определить, с каким контейнером необходимо взаимодействовать. Также идентификатор контейнера может использоваться для удаления контейнера, извлечения его журналов, изменения его конфигурации и других операций, предоставляемых Docker API.

В целом, атрибут `id` является ключевым компонентом для идентификации и управления контейнерами в системе Docker, обеспечивая их уникальность и возможность выполнения различных операций через Docker API.

Атрибут `status` является важным компонентом, отображающим текущее состояние контейнера. Он предоставляет информацию о том, запущен ли контейнер, завершен ли его процесс, или он приостановлен. Значение атрибута `status` может изменяться в зависимости от действий, выполняемых с контейнером, и может принимать различные значения, включая:

– running (запущен): это означает, что контейнер активен и его процесс выполняется в данный момент. Контейнер находится в рабочем состоянии и готов обрабатывать запросы или выполнять другие задачи.

– exited (завершен): это состояние указывает на то, что процесс в контейнере завершился, и контейнер больше не активен. Это может произойти после завершения выполнения задачи или программы в контейнере.

– paused (приостановлен): контейнер находится в приостановленном состоянии, что означает временную приостановку его выполнения. Это может быть вызвано, например, для снижения нагрузки на систему или для временного приостановления работы контейнера без его полного останова.

Атрибут `status` обеспечивает быстрый и удобный способ определения текущего состояния контейнера без необходимости выполнять дополнительные запросы к Docker API. Он играет важную роль в мониторинге и управлении контейнерами, позволяя операторам и разработчикам быстро оценить состояние контейнеров и принять соответствующие действия в зависимости от их текущего состояния.

Атрибут `url` в контексте контейнера представляет собой URL-адрес, который может содержать ссылку на контейнер, если такая информация доступна или применима в рамках проекта. URL-адрес контейнера может играть важную роль в управлении и мониторинге контейнеров, особенно в средах, где применяется микросервисная архитектура или веб-приложения.

Основная функциональность атрибута `url` включает:

– быстрый доступ к сервисам контейнера: предоставление URL-адреса может обеспечить быстрый доступ к сервисам, запущенным в контейнере. Это может быть особенно полезно при работе с веб-приложениями или микросервисами, где URL-адрес используется для взаимодействия с контейнером.

– мониторинг и диагностика: URL-адрес контейнера может использоваться для мониторинга его состояния или для диагностики проблем, связанных с его работой. Путем доступа к контейнеру через URL-адрес можно получить доступ к логам, метрикам или интерфейсам для анализа его работы.

– интеграция с другими сервисами: URL-адрес контейнера может быть интегрирован в другие сервисы или инструменты для автоматизации или управления им. Например, его можно использовать в скриптах или конфигурационных файлах для автоматического обновления или масштабирования контейнера.

Однако следует отметить, что в некоторых случаях атрибут `url` может быть пустым или неиспользуемым, особенно если URL-адрес не применим к конкретному контейнеру или если его использование не предусмотрено в контексте проекта. В таких случаях атрибут может быть опущен или заполнен значением по умолчанию.

Атрибуты класса Container предоставляют основную информацию о контейнере, необходимую для его управления и мониторинга в рамках приложения. Они обеспечивают удобный доступ к основным характеристикам контейнера, что делает класс Container ключевым компонентом взаимодействия с контейнерами Docker.

3.2.2 ContainerCreate

ContainerCreate – это структура данных, предназначенная для передачи параметров, необходимых для создания нового контейнера в среде Docker. Она играет важную роль в процессе управления контейнерами и обеспечивает возможность настройки различных аспектов создаваемого контейнера

Особенности ContainerCreate:

– параметры создания контейнера: класс ContainerCreate содержит информацию о параметрах, необходимых для создания контейнера. Сюда входят данные, такие как образ контейнера, команда для выполнения внутри контейнера, переменные окружения, порты и другие настройки;

– использование с методом create_container: обычно ContainerCreate используется вместе с методом create_container класса ContainerService. Метод create_container принимает экземпляр ContainerCreate в качестве параметра и использует предоставленные данные для создания нового контейнера;

– гибкость настройки: благодаря ContainerCreate можно гибко настраивать параметры создаваемого контейнера в зависимости от требований проекта. Это позволяет управлять различными аспектами контейнера, такими как его окружение, ресурсы и поведение;

– удобство использования: использование ContainerCreate делает процесс создания контейнера более удобным и понятным. Все необходимые параметры указываются явно в виде атрибутов структуры данных, что облегчает понимание и поддержку кода.

Атрибуты класса ContainerCreate следующие:

– `env`;

– `image`;

– `command`;

– `labels`.

Атрибут `env` в классе ContainerCreate является важным компонентом при создании контейнера в среде Docker. Этот атрибут позволяет задавать переменные окружения, которые будут доступны внутри создаваемого контейнера. Переменные окружения играют ключевую роль в настройке и конфигурации приложений, работающих в контейнерах, и предоставляют способ передачи важных параметров и конфигураций.

Подробнее о атрибуте `env`:

– словарь переменных окружения: атрибут `env` представляет собой словарь, где ключами являются имена переменных окружения, а значениями – соответствующие значения переменных;

– использование переменных окружения: в контейнерах переменные окружения могут использоваться для различных целей, таких как указание параметров конфигурации приложения, настройка окружения выполнения, передача секретных данных (например, паролей или ключей API) и т.д.;

– примеры переменных окружения: переменная окружения `DATABASE_URL` может содержать URL-адрес или строку подключения к базе данных, которая будет использоваться приложением для соединения с базой данных. Переменная окружения `DEBUG` может определять, включен ли режим отладки в приложении, что может влиять на уровень журналирования или поведение приложения;

– гибкость настройки: использование переменных окружения обеспечивает гибкость настройки приложений в контейнерах, так как позволяет изменять их поведение без изменения самого приложения. Это удобно для конфигурации приложений в различных средах (например, разработка, тестирование, производство) или для переноса приложений между разными инфраструктурами.

Атрибут `image` в классе ContainerCreate играет важную роль при создании контейнера в Docker. Он определяет, какой именно образ Docker будет использоваться для создания контейнера. Образ Docker представляет собой шаблон, содержащий файловую систему с установленными программами, библиотеками и другими зависимостями, необходимыми для работы приложения.

Вот более подробное описание атрибута `image`:

– указание на образ Docker: атрибут `image` является строкой, которая указывает на конкретный образ Docker. Образ обычно идентифицируется своим именем вместе с тегом (например, nginx:latest), который указывает на конкретную версию образа;

– определение файловой системы и программного обеспечения: образ Docker определяет файловую систему и установленное программное обеспечение, которые будут доступны в контейнере. Все зависимости и файлы, необходимые для работы приложения, должны быть настроены в этом образе;

–версионирование образов: образы Docker могут иметь различные версии, обозначаемые тегами. Например, `nginx:latest` указывает на последнюю версию образа `Nginx`, в то время как nginx:1.19 указывает на конкретную версию 1.19;

– использование образов из репозиториев: образы Docker могут быть получены из публичных или частных репозиториев, таких как Docker Hub, где они хранятся и распространяются. Это облегчает доступ к готовым образам для развертывания приложений.

Атрибут `command` содержит команду, которая будет выполнена внутри контейнера при его запуске. Команда определяет, какое приложение или процесс должно быть запущено в контейнере. Обычно это представляет собой строку или список строк, где каждая строка представляет отдельную команду и ее аргументы.

Атрибут `labels` в классе ContainerCreate играет важную роль в организации и идентификации контейнеров в Docker-среде. Метки представляют собой ключ-значение пары, которые могут быть присвоены контейнеру при его создании. Эти метки могут быть использованы для различных целей, таких как организация контейнеров в группы, идентификация контейнеров с определенными свойствами или применение фильтров при выполнении операций на контейнерах.

Вот более подробное описание атрибута `labels`:

– уникальная идентификация: метки могут быть использованы для уникальной идентификации контейнера или группы контейнеров. Например, метка `app` с указанием имени приложения может быть присвоена контейнеру для его идентификации в рамках приложения;

– организация контейнеров: метки позволяют организовать контейнеры в логические группы или категории. Например, метка `role` с указанием роли контейнера (например, frontend, backend, database) помогает классифицировать контейнеры по их функциональности;

– фильтрация и поиск: метки могут использоваться для фильтрации контейнеров при выполнении операций на них. Например, при использовании Docker CLI или Docker API можно выполнить операции только на контейнерах с определенными метками, что облегчает управление контейнерами в больших окружениях;

– метаданные и конфигурация: метки могут использоваться для хранения метаданных или конфигурационных параметров, связанных с контейнером. Например, метки могут содержать версию приложения, настройки сети или другие параметры, которые могут быть полезны при управлении контейнером.

3.2.3 ContainerBase

Класс ContainerBase служит в качестве базового класса для создания контейнеров. Атрибуты класса следующие:

– `image` – строка, содержащая имя образа;

– `command` – строка или список строк, содержащих команду;

– `labels` – словарь меток, которые могут быть присвоены контейнеру.

Класс ContainerBase предоставляет базовую структуру для определения этих атрибутов, которые могут быть настроены в подклассах или при создании экземпляров объектов этого класса для создания контейнеров с различными конфигурациями и параметрами.

3.2.4 Scale

Класс Scale представляет собой абстракцию, описывающую масштабирование приложения или сервиса. Атрибуты класса следующие:

– `service_name`;

– `replicas`.

Атрибут `service_name` – это ключевой параметр в структуре проекта, который используется для идентификации конкретного сервиса или компонента в рамках системы. В контексте проекта, вероятно, это используется для указания имени сервиса, который будет масштабирован или управляем данным классом.

Вот более подробное описание атрибута `service_name`:

– идентификация сервиса: атрибут `service_name` предназначен для идентификации конкретного сервиса в системе. Каждый сервис может иметь уникальное имя, которое используется для его обозначения и обращения к нему в рамках проекта;

– управление масштабированием: В контексте класса ScaleService, атрибут `service_name` вероятно используется для указания имени сервиса, который требуется масштабировать. Это позволяет точно определить, какой именно сервис должен быть масштабирован в ответ на изменения в нагрузке или других факторах;

– ключевой параметр методов: атрибут service_name вероятно является ключевым параметром для методов класса ScaleService, таких как `scale_up()` или `scale_down()`. Он используется для указания конкретного сервиса, который требуется масштабировать, и определения количества экземпляров, которые должны быть добавлены или удалены;

– уникальность идентификатора: имя сервиса должно быть уникальным в рамках системы, чтобы избежать конфликтов и путаницы при обращении к сервисам. При выборе имени сервиса важно учитывать его уникальность и понятность для других членов команды;

– конфигурационный параметр: атрибут `service_name` может также использоваться в качестве конфигурационного параметра для других компонентов системы, которые могут взаимодействовать с сервисом, таких как мониторинг, журналирование или автоматическое масштабирование.

Атрибут `replicas` – это параметр, определяющий количество экземпляров сервиса, которые требуется создать или уничтожить при масштабировании. В контексте класса ScaleService, этот атрибут играет ключевую роль при увеличении или уменьшении количества контейнеров, обслуживающих конкретный сервис.