Детализированная модель построения системы обработки данных с использованием технологий виртуализации

Селезнёв Александр Игоревич, ассистент;
Селезнёв Игорь Львович, кандидат технических наук, доцент
Белорусский государственный университет информатики и радиоэлектроники (г. Минск)

В статье рассматривается построение детализированной системы обработки данных. Сформулированы основные этапы проектирования систем обработки данных с использованием технологий виртуализации, произведен выбор необходимой технологии виртуализации, системной архитектуры и спроектирована детализированная модель обработки данных.

Ключевые слова: *система обработки данных, контейнеры, микросервисная архитектура, системы оркестрации, виртуализация.*

Системы обработки данных (СОД) являются важным технологическим звеном в современном мире. В работе [1] авторами было рассмотрено построение модели обобщенной системы обработки данных. Учитывая особенности построения подобных систем важно выбрать оптимальные технологии для гибкой и эффективной обработки возрастающих массивов данных, что и является целью данной публикации. Одним из распространённых вариантов проектирования систем обработки данных является использование технологий виртуализации, позволяющие разделить ресурсы физического сервера или серверных кластеров между многими пользователями и приложениями.

Для проектирования детализированной модели построения СОД с использованием технологии виртуализации определены следующие основные задачи:

– исследование систем и основных этапов обработки данных;

– исследование и анализ основных типов виртуализации;

– исследование системных архитектур, необходимых для построения систем обработки информации на основе технологий виртуализации;

– проектирование детализированной системы обработки данных.

На основе проведенных исследований [1–3] было выдвинуто ключевое требование к проектируемой системе — способность решать задачи преобразования различных данных в требуемые выходные с обеспечением надлежащей безопасности, быстрой интеграции используемых средств и обновлений системы. Для этого проектируемая СОД должна удовлетворять следующим критериям:

1. Производительность и надежность. СОД должна быть высокопроизводительной и надежной системой, способной выдерживать большие нагрузки и работать без потери данных.

2. Гибкость. Разрабатываема система должна обладать высокой степенью интеграции и гибкостью, каждая из её частей должна быть отдельным самостоятельным блоком, допускающим, при необходимости, внесение изменений без остановки и перезагрузки всей системы.

3. Технологичность. При разработке СОД должны применяться современные технологии.

4. Безопасность. Все входные и выходные данные должны быть защищены от взлома и внешнего влияния.

Для разработки СОД, удовлетворяющей данным требованиям, необходимо решить следующие задачи:

1. Выбор и обоснование необходимых технологий виртуализации.

2. Выбор и обоснование системной архитектуры.

3. Детализация характеристик элементов выбранной технологии виртуализации.

4. Построение уточненной структуры обработки данных.

Выбор и обоснование необходимых технологий виртуализации

Для проектирования СОД были рассмотрены две популярные технологии виртуализации — виртуальные

машины и контейнеры; произведен анализ и сравнение данных технологий по следующим критериям [4]:

1. Развертывание. Оба решения могут быть развернуты при помощи загрузки файлового образа. Виртуальную машину (ВМ) также можно развернуть вручную.

2. Изолированность. Виртуальные машины имеют полноценную логическую изоляцию. Контейнеры используют единое ядро операционной системы (ОС), поэтому они не изолированы на 100%.

3. Производительность. Оба решения обеспечивают высокую производительность, однако ВМ имеют небольшое преимущество. Связано это с логическим отделением каждой машины от других, то есть при работе у элементов не возникает конкуренция за используемые ресурсы.

4. Отказоустойчивость. Обе технологии виртуализации обеспечивают отказоустойчивость системы. Однако контейнеры первоначально не имеют статического состояния, поэтому при сбое позволяют избежать потери данных. ВМ с этой точки зрения потенциально опаснее.

5. Сетевое взаимодействие. Представленные технологии позволяют компонентам взаимодействовать как в рамках одного компьютера, так и в системе, объединенной сетью. Однако контейнеры способны нагрузить пропускную способность сети гораздо быстрее.

6. Безопасность. Виртуальные машины используют собственные операционные системы и приложения, что делает применение каждой системы более безопасным. Контейнеры более подвержены влиянию проблем ОС, так как используют общую систему.

7. Масштабируемость. Контейнеры масштабируются проще, так как имеют меньший размер и требуют меньше ресурсов. ВМ больше подходят для больших постоянных нагрузок.

Ввиду того, что одним из критериев проектируемой СОД выступает требование гибкости, крайне важна масштабируемость и отказоустойчивость, что предопределяет выбор в качестве основной технологии виртуализации использование контейнеров.

Выбор и обоснование системной архитектуры

Исходя из требований к разрабатываемой системе произведен анализ и выбор наиболее подходящей системной архитектуры. Были рассмотрены монолитная архитектура и микросервисная архитектура (МА). В качестве критериев сравнения архитектур используются следующие [5]:

1. Структура. Монолитные приложения используют единую кодовую базу и одну платформу, в то время как микросервисы состоят из множества слабосвязанных сервисов. Микросервисы не зависят от платформы и стека технологий. Каждый микросервис может быть разработан с использованием различных языков программирования, баз данных и технологий, например, один микросервис может быть создан на Node.js с использованием базы данных NoSQL (Not only SQL, не только SQL), а другой — на Java с использованием реляционной базы данных.

В отличие от этого, монолитные приложения как правило используют единый стек технологий для всего приложения, что накладывает существенные ограничения при разработке и модернизации приложения.

2. Масштабируемость. Монолитные приложения масштабируются путем развертывания всего приложения, в то время как микросервисы позволяют масштабировать отдельные сервисы, что дает возможность эффективной работы при нагрузке на один из элементов этой архитектуры.

В монолитной архитектуре масштабирование обычно предполагает развертывание всего приложения. Поэтому, если одна часть приложения требует больше ресурсов, масштабировать нужно весь монолит, что может привести к неэффективному распределению ресурсов.

3. Разработка. На начальном этапе разработки использование монолитной архитектуры позволяет ускорить разработку проекта в большей степени, чем МА — но в дальнейшем обеспечивает меньшую гибкость.

При использовании микросервисов команды разработчиков могут сосредоточиться на отдельных сервисах. Например, одна команда может отвечать за службу каталога товаров, а другая — за аутентификацию пользователей. Это позволяет использовать специализированные знания и ускорить цикл разработки в каждой команде.

В монолитной архитектуре разные команды могут работать над разными частями монолита, но при этом возрастает риск конфликтов и проблем с координацией, поскольку изменения в одной части монолита могут повлиять на другие.

4. Развертывание. Одним из ключевых преимуществ микросервисов является их гибкость в развертывании. Каждый сервис в архитектуре микросервисов может быть развернут независимо. Это означает, что когда приходит время выпустить новую функцию или обновить определенный функционал в программном обеспечении, то не нужно осуществлять развертывание всего приложения, вместо этого можно развернуть только те службы, которые были изменены.

В отличие от этого, монолитные приложения развертываются как единое целое. Когда нужно выпустить новую функцию или обновление, то необходимо перезапустить весь монолит. Это может быть более рискованным и сложным процессом, особенно если изменения существенны.

Возможность развертывания только необходимых компонентов в архитектуре микросервисов особенно выгодна в среде гибкой разработки, где быстрые и частые выпуски новых версий являются нормой. Это позволяет командам разработчиков быстро выполнять итерации и более эффективно реагировать на отзывы пользователей, что в конечном итоге приводит к созданию более динамичной и отзывчивой экосистемы программного обеспечения.

Таким образом, обе архитектуры имеют свои проблемы. Монолитные приложения могут стать громоздкими по

мере роста, а микросервисы усложняют управление межсервисным взаимодействием. Для проектируемой СОД выбирается микросервисная архитектура, так как она отвечает поставленным требованиям в большей мере, чем монолитная архитектура.

Детализация характеристик элементов технологии контейнеризации

Технология контейнеризации предполагает собой использование контейнеров. Несмотря на то, что в настоящее время одним из популярных видов контейнеров является Docker, перед выбором в качестве основного элемента технологии контейнеризации для проектирования СОД важно произвести сравнение и с другим распространённым контейнером — Linux Containers (LXC). Анализ и сравнение данных видов контейнеров производится по следующим критериям:

1. Использование хост-машины. И контейнеры LXC, и контейнеры Docker взаимодействуют с ядром хост-машины, поэтому для понимания особенностей этих технологий важно знать, каким образом в ОС Linux организована работа с памятью. В этой ОС память делится на пространство ядра, предназначенное для ядра операционной системы, отвечающего за управление процессором, памятью и устройствами, и пространство пользователя (рис. 1).

Доступ к пространству ядра для пользовательских процессов возможен только с помощью системных вызовов. Системные вызовы — это запросы активного процесса в Unix-подобной операционной системе на услуги, выполняемые ядром, такие как ввод/вывод (I/O) или создание процесса.

Контейнеры LXC используют возможности ядра Linux для создания изолированных процессов и файловых систем. Например, в Linux используя системную команду для вывода запущенных процессов на хосте будет выведен список активных процессов, включая контейнер LXC (который с точки зрения хоста также является процессом). Результатом выполнения этой команды при запуске из контейнера будет список процессов, запущенных только внутри контейнера. Для достижения этой возможности LXC использует пространство имен ядра (kernel namespaces).

Ранние версии Docker представляли собой усовершенствованную версию LXC, а с 2015 года Docker уже использует собственную библиотеку libcontainer в качестве среды выполнения (средства взаимодействия с Linux), абстрагирующую виртуализационные возможности ядра Linux.

Структурные отличия Docker от LXC контейнеров на уровне использования хоста изображены на примере приложения «ReadApp», изображенного на рисунке 2.

Docker предлагает абстракцию для специфических настроек машины, таких как сеть, хранилище, протоколирование и тому подобное. Эта абстракция является частью программного ядра Docker (в оригинале Docker Engine [6]) и делает контейнеры Docker более переносимыми, поскольку они меньше зависят от базовой физической машины. Особенностью контейнеров Docker является то, что они предназначены для запуска одного процесса.

2. Простота и гибкость. Контейнеры Linux более гибки по своей конструкции. Они больше похожи на виртуальные машины: их настройка и установка обладает таким же многообразием, как и ВМ. Для создания виртуальной среды LXC можно использовать такие возможности ядра, как chroot, cgroups и namespaces. Эти механизмы ядра помогают контролировать использование ресурсов и видимость процессов для остальной системы.

Контейнеры Docker изначально были созданы на основе проекта LXC, однако концептуально они сильно отличаются: контейнеры Docker были разработаны специально для приложений микросервисов, что разнит их от ВМ. Docker прост в использовании для разработчиков: абстракция сетей, хранилищ и протоколирование не требует предварительных знаний Linux. Именно простота, которую Docker предлагает разработчикам, сделала его таким популярным.

3. Производительность и скорость работы. LXC характеризуется быстрой загрузкой по сравнению с виртуальной машиной — этому типу контейнеров не нужно

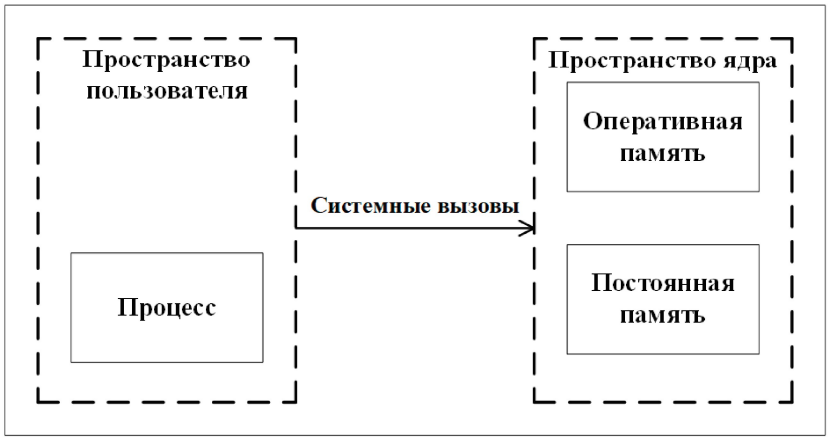

Рис. 1. **Структура памяти ОС Linux**

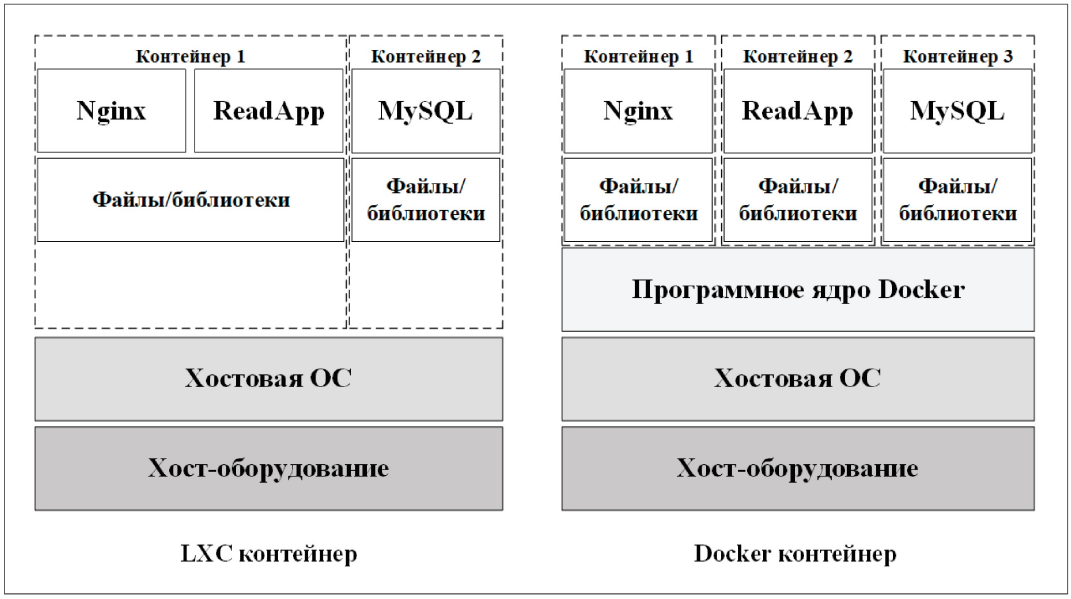

Рис. 2. **Пример приложения «ReadApp» на контейнерах LXC и Docker**

упаковывать целую ОС и полную конфигурацию машины с сетевыми интерфейсами, виртуальными процессорами и жестким диском.

Контейнеры Docker имеют небольшой размер исполняемых файлов, что значительно повышает их скорость. Docker добавляет дополнительный слой для абстрагирования хранилища и сети, но в большинстве случаев это не вызывает значительных проблем с производительностью. Создание и запуск контейнера Docker в сравнении с ВМ происходит значительно быстрее. Эти действия требуют еще меньших затрат времени, если Docker работает на существующей ОС, которая уже инициализирована.

По умолчанию Docker поддерживает многослойные контейнеры. Это означает, что результирующий контейнер представляет собой последовательную комбинацию изменений, внесенных в файловую систему. Слои можно загружать параллельно, что может дать преимущество в скорости запуска множества контейнерных приложений на нескольких машинах одновременно.

Еще одним преимуществом Docker является возможность создания бездистрибутивных образов Docker. В основе таких образов лежит необходимый минимум объектов: приложение, скомпилированный код и требуемая привязка. В отличие от этого образы LXC, как правило, копируют дистрибутивы Linux целиком, например такие как Ubuntu, Debian или Alpine. Docker в большей степени предназначен для упаковки отдельных приложений, что дает преимущество при увеличении количества контейнеров.

Разница в производительности между LXC и Docker несущественна. Оба обеспечивают быстрое время загрузки. Загрузка образа LXC может быть медленнее, чем бездистрибутивные образы Docker.

4. Безопасность. LXC оснащен такими инструментами управления безопасностью, как групповые политики и профиль AppArmor по умолчанию, чтобы защитить хост от случайного злоупотребления привилегиями внутри контейнера.

Docker отделяет операционную систему от выполняемых на ней служб, чтобы обеспечить безопасность рабочих нагрузок, но тот факт, что он запускается от имени root (максимальные привилегии по доступу и управлению ОС), может увеличить уязвимость для воздействия вредоносного ПО. Это связано с тем, что демон Docker, который управляет такими объектами, как сети, контейнеры, образы и тома, а также отвечает на запросы Docker API, запускается на хост-машине с правами root. Вследствие этого разработчикам необходимо проверять установки Docker для выявления потенциальных уязвимостей.

LXC имеет высокий уровень защиты. Он предоставляет функции безопасности, включая поддержку возможностей Linux, чтобы помочь сохранить контроль над всей контейнерной средой и размещенными приложениями. Подход Docker к хранению различных компонентов приложений в отдельных контейнерах является его достоинством, однако накладывает свои ограничения по безопасности для сложных приложений.

5. Простота использования. LXC запускает стандартный блок операционной системы для каждого контейнера — приложения размещаются в ОС Linux. Таким образом, переход на контейнеры LXC с «голого железа» или серверов виртуальных машин более управляем, чем переход на контейнеры Docker.

С другой стороны, Docker упрощает работу с контейнерами и запуск программ. С помощью Docker становиться возможным упаковывать контейнеры и отправлять их для использования на другие машины. Поскольку Docker является контейнерной платформой с открытым исходным кодом, это облегчает работу с ним.

6. Масштабируемость. LXC менее масштабируема по сравнению с Docker. Его образы не такие «легкие», как об-

разы Docker. Однако образы LXC более легковесны, чем образы физических или виртуальных машин. Это делает его удобным для отложенной инициализации и автоматического масштабирования. Кроме того, тот факт, что с помощью LXC можно реализовать легковесные виртуальные машины без гипервизора, делает контейнеры Linux масштабируемым вариантом.

Docker позволяет разделять функциональность приложений на отдельные контейнеры. Например, можно запустить базу данных Oracle в одном контейнере, а сервер Cassandra и приложение ASP.NET — в других, отдельных контейнерах. Затем можно соединить эти три контейнера вместе и создать приложение, обеспечивающее независимое масштабирование компонентов.

7. Инструментарий. Инструментарий LXC поддерживает выполнение ряда команд, обеспечивающих управление такими задачами, как создание, запуск и удаление контейнеров LXC. Этот инструментарий также позволяет повторно использовать сценарии автоматизации, схожие со сценариями на ВМ в VirtualBox или других виртуализированных производственных средах. Благодаря такой гибкости можно легко перенести приложения с традиционного Linux-сервера в Linux-контейнер.

Интерфейс командной строки (Command Line Interface, CLI) Docker — это «сердце» инструментария Docker. Он обеспечивает контроль над контейнерами, позволяя выводить список и управлять образами. Также можно использовать реестр Docker для доступа и распространения образов часто используемых приложений.

И LXC, и инструментарий Docker предоставляют важные функции для повышения удобства разработчиков. Инструментарий LXC отлично подходит для переноса приложений с традиционного Linux-сервера в Linux-контейнер, инструментарий Docker, в свою очередь, больше подходит для разработки.

Из вышесказанного понятно, что LXC больше подходит для Linux-приложений, которые нуждаются в Linux-окружении и для серверных приложений. Docker — комплексное решение для распространения приложений, поддерживающее изоляцию между рабочей нагрузкой и воспроизводимой средой. В следствии этого для проектирования СОД были выбраны именно Docker контейнеры.

Для того, чтобы эффективно производить управление контейнерами, необходимо выбрать наиболее подходящую технологию контейнеризации. С этой целью далее рассматривается архитектура, анализируются достоинства и недостатки таких популярных технологий оркестрации, как Kubernetes и Docker Swarm, а затем производится сравнение и выбор технологии оркестрации.

Архитектура Kubernetes построена на принципах распределенной системы, пример которой представлен на рисунке 3.

Основными структурными компонентами архитектуры Kubernetes являются: мастер-узел, рабочие узлы и поды (pods) [7].

Мастер-узел (Master Node) — это главный управляющий узел Kubernetes. Он отвечает за принятие решений о состоянии и размещении контейнеров, а также за управление всеми узлами в кластере. Мастер-узел состоит из следующих элементов:

1. API сервера — интерфейс для взаимодействия с Kubernetes, который принимает запросы на управление кластером.

2. Планировщик (Scheduler) — это компонент, отвечающий за распределение контейнеров на рабочие узлы в зависимости от доступных ресурсов и требований приложения. Он обеспечивает балансировку нагрузки и оптимальное использование ресурсов кластера.

3. Контроллеры (Controllers) — это компоненты, отвечающие за отслеживание текущего состояния кластера. Их задача — приведение кластера к требуемому состоянию. В Kubernetes есть множество встроенных контроллеров, которые работают внутри kube-controller-manager.

4. База данных etcd — это база данных (БД), которая используется для хранения состояния кластера, включая конфигурацию, состояние узлов и подов. Она обеспечивает согласованность данных в кластере.

Рабочие узлы (Worker Nodes) — это узлы, в которых выполняются контейнеры. Они включают в себя следующие компоненты:

1. Поды — это минимальные единицы развертывания в Kubernetes. Под представляет собой один или несколько контейнеров, работающих вместе на одном рабочем узле. У них есть общая сеть и хранилище. Поды могут содержать несколько контейнеров, которые работают в тесной связке и могут обмениваться данными через общее пространство имен.

2. Среда выполнения, например Docker, отвечающая за получение (загрузку) образа контейнера из реестра, распаковку контейнера и запуск приложения.

3. Kubelet — агент Kubernetes, который управляет жизненным циклом подов на рабочем узле. Он запускает и останавливает контейнеры в соответствии с указаниями от мастер-узла, а также отслеживает их состояние.

4. Kube-Proxy — это компонент, обеспечивающий сетевую связность между подами и другими сервисами в кластере. Он выполняет функцию балансировщика нагрузки и маршрутизатора для обеспечения сетевой доступности приложений.

Kubernetes обладает следующими достоинствами [8]:

1. Большое сообщество разработчиков с открытым исходным кодом, поддерживаемое компанией Google.

2. Поддержка всех наиболее популярных ОС, таких как Windows, Linux и т.д.

3. Поддержка и управление большими архитектурами и сложными рабочими нагрузками.

4. Kubernetes полностью автоматизирована и обладает способностью к самовосстановлению с возможностью автоматического масштабирования.

5. Встроенный мониторинг и широкий спектр интеграций.

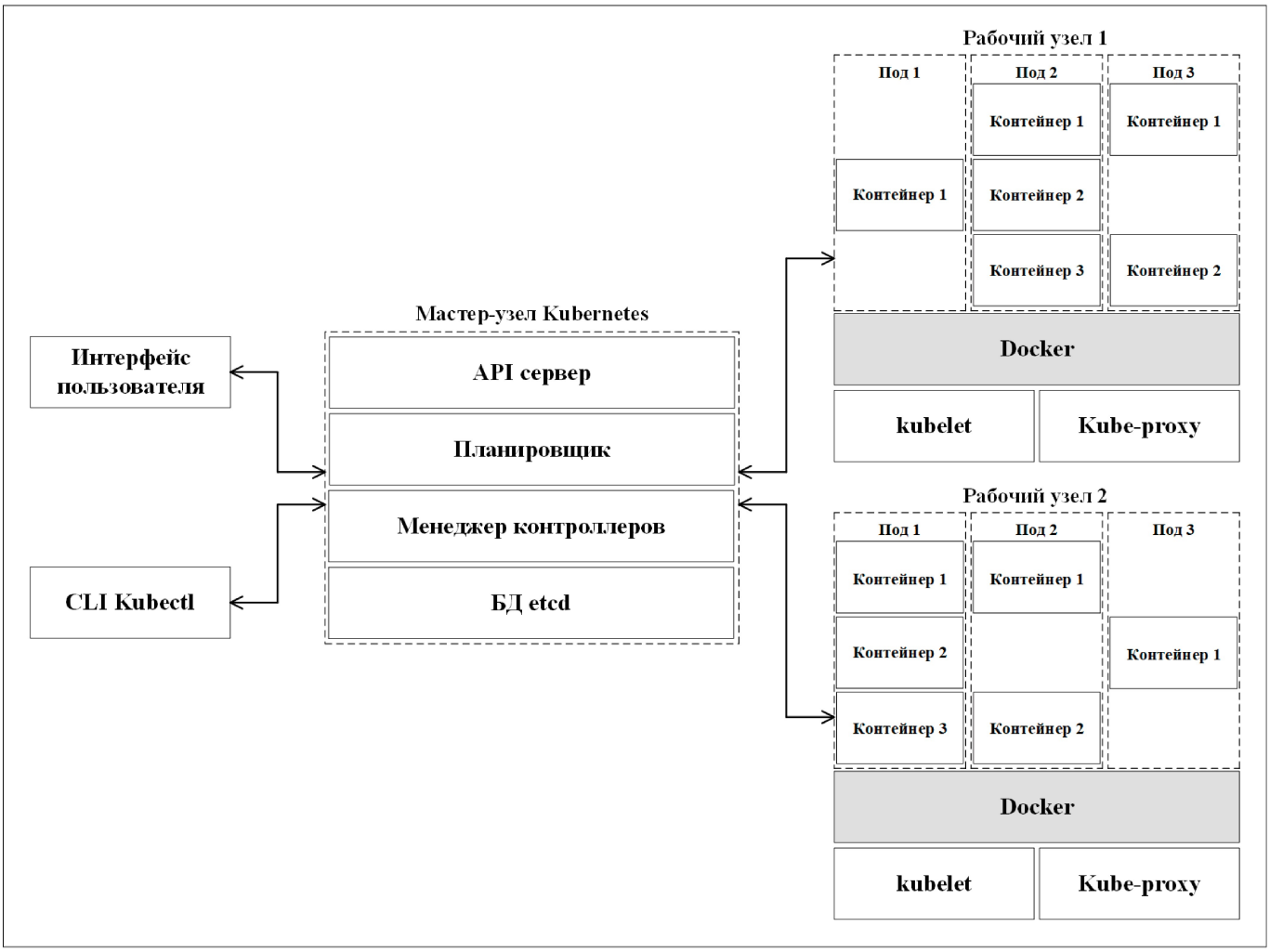

Рис. 3. **Пример архитектуры кластера Kubernetes**

6. Доступность у всех крупных международных облачных провайдеров: Google, Azure и Amazon Web Services (AWS).

Благодаря широкой поддержке сообщества и способности справляться даже с самыми сложными сценариями развертывания Kubernetes наиболее подходит команде разработчиков предприятия, управляющих приложениями на базе микросервисов.

Недостатки технологии Kubernetes:

– сложный процесс установки и обучения;

– требование к установке отдельных инструментов CLI и изучения каждого из них.

В некоторых ситуациях Kubernetes может оказаться слишком сложной и привести к снижению производительности работы приложений.

Архитектура системы оркестрации Docker Swarm представляет собой систему взаимодействия swarm менеджеров (swarm node) и swarm рабочих узлов (swarm worker node), пример которой изображен на рисунке 4 [9, 10].

Эта система оркестрации состоит из следующих компонентов:

1. Задача. Задачу можно понимать как комбинацию одного контейнера Docker и команд, определяющих, как этот контейнер будет запущен и как он будет работать.

2. Сервис, состоящий из одной или нескольких задач.

3. Группа распределенного консенсуса RAFT (Raft Consensus Group) состоит из внутренней и внешней распределенной базы данных состояний и рабочих узлов. RAFT — это алгоритм консенсуса для управления реплицированным журналом [11]. Каждая распределенная система должна реплицировать состояние, чтобы получить метку; почти каждая устойчивая распределенная система реализует протокол RAFT.

Консенсус означает, что большинство в группе согласны с каким-либо утверждением. В терминах RAFT или распределенных систем консенсус подразумевает, что большинство узлов в кластере согласны с определенным изменением в системе. Это согласие большинства гарантирует, что в случае изменения лидера новым лидером будет тот, кто был частью большинства.

4. Внутреннее распределенное хранилище состояния (Internal Distributed State Store) предназначено для хранения данных о состоянии контейнерного кластера (среды, в которой работают контейнеры) в формате «ключ-значение».

5. Менеджеры Docker Swarm представляют собой управляющие узлы и состоят из следующих частей:

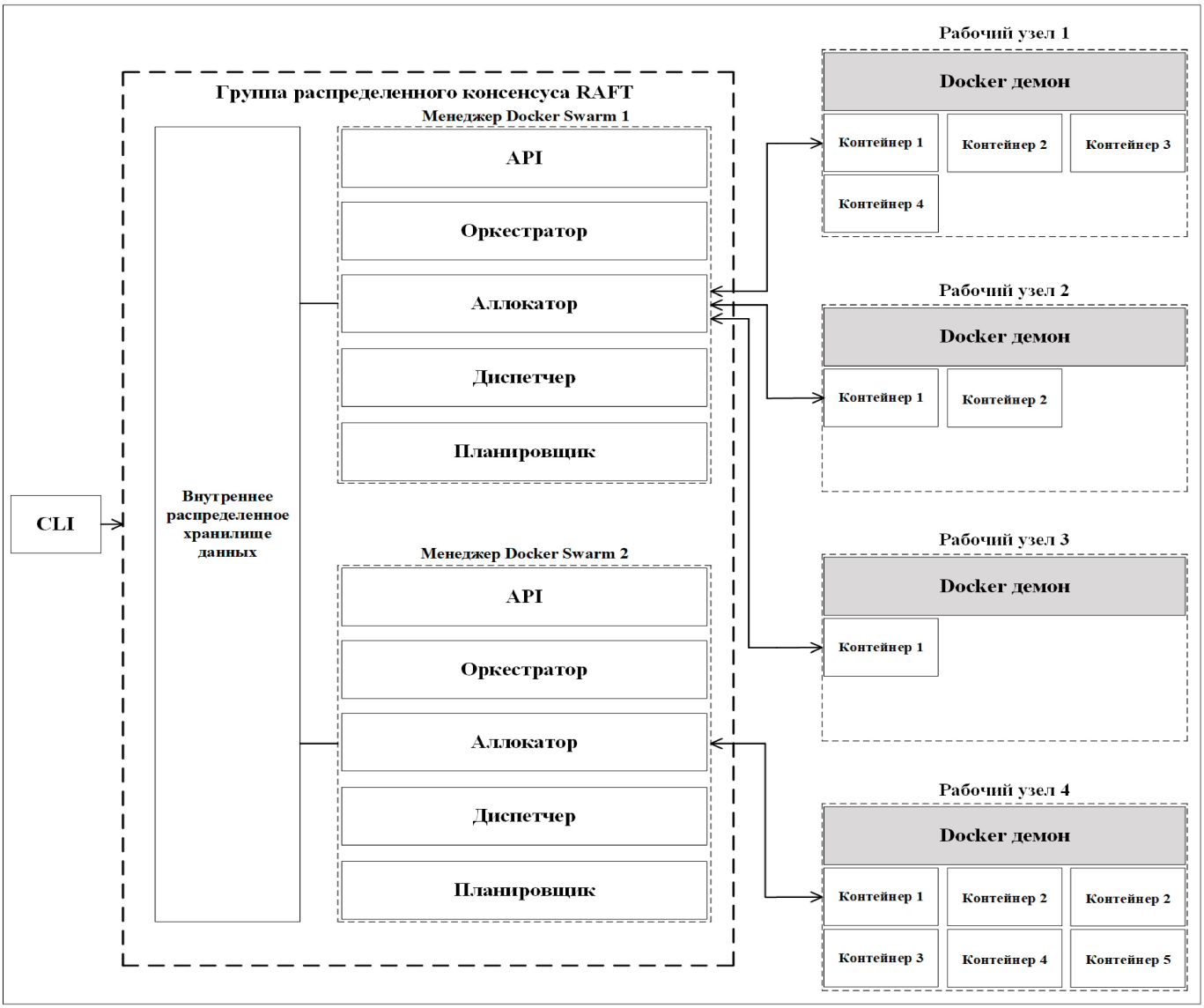

Рис. 4. **Пример архитектуры системы оркестрации Docker Swarm**

– API, посредством которого можно получать команды от пользователей и создавать новые сервисы на основе параметров, заданных во входящих командах;

– оркестратор служит для приема определения (defiition) сервиса и создания задач;

– аллокатор предназначен для выделения и назначения IP-адресов задачам;

– диспетчер осуществляет контроль над рабочими узлами;

– планировщик составляет расписание задач и назначает их рабочим узлам.

6. Рабочие узлы получают задания непосредственно от управляющего узла и затем выполняют их. Эти узлы отправляют текущий статус выполненных заданий на управляющий узел; а также в дополнение к этому статусу, они отправляют информацию о своем собственном состоянии управляющему узлу, информируя его о том, что они все еще могут принять поставленные задачи и выполнить их.

Достоинства Docker Swarm [12]:

1. Управление контейнерными кластерами с помощью среды Docker. Контейнерные кластеры можно создавать и управлять ими непосредственно через командную строку Docker (CLI), не требуется дополнительное стороннее программное обеспечение для оркестрации.

2. Децентрализованная структура построения. Вместо того чтобы заранее определять типы отдельных узлов (управляющие, исполняющие) в процессе развертывания контейнерного кластера, различие этих узлов будет проявляться только во время работы контейнеров; то есть весь контейнерный кластер может быть собран из одного образа, независимо от того, какие типы узлов будут в кластере.

3. Декларативная модель проектирования. Docker Swarm предоставляет собой декларативный подход к определению сервисов, который позволяет определить желаемое состояние и тип различных сервисов. Например, эта модель позволяет создать приложение, со-

стоящее из очередей конечных веб-сервисов и нескольких внутренних сервисов (очереди сообщений, база данных).

4. Масштабирование. Для каждого сервиса можно определить количество выполняемых задач. При масштабировании Docker Swarm может автоматически добавлять или удалять задания по мере необходимости, поддерживая контейнерное окружение в требуемом состоянии.

5. Сохранение требуемого состояния контейнерных кластеров. Узел управления постоянно отслеживает текущее состояние и, при необходимости, исправляет возникшие ошибки, которые привели бы к некорректному состояния. Примером может служить ситуация, когда создается десять копий одного из контейнеров, две из которых запускаются на исполняющем узле, и узел неожиданно выходит из строя. В этом случае главный узел создает две новые копии контейнера, которые заменяют две некорректно работающие копии и назначает их на исполняющие узлы, которые исправны и способны принять эти копии.

6. Создание сетей в кластере контейнеров. Docker Swarm позволяет создавать так называемые оверлейные сети для сервисов. При инициализации или обновлении приложения узел управления автоматически назначает IP-адреса контейнерам, расположенным в такой сети.

7. Балансировка нагрузки. Порты отдельных сервисов могут быть подвергнуты внешней балансировке нагрузки. Внутреннее распределение нагрузки, в свою очередь, обеспечивается за счет распределения контейнеров в кластере между отдельными узлами.

8. Безопасность. Каждый узел в кластере может выполнять взаимную аутентификацию и шифрование для защиты связи между собой и другими узлами в кластере. Эти операции защищены с помощью сертификатов TLS, причем можно использовать самоподписанные сертификаты в качестве корневых, но также можно применять сертификаты, подписанные другим центром сертификации (Certification authority, CA).

9. Создание точек восстановления для обновлений. Обновления версий служб могут выполняться последовательно, при этом для каждого обновления создается точка восстановления. Если обновление сервиса не удалось, можно вернуться к предыдущей версии сервиса.

10. Контроль развертывания сервисов. Управляющий узел может контролировать задержку развертывания сервисов на отдельных узлах.

Недостатки технологии Docker Swarm:

– легковесность и обязательная привязка к API Docker, что ограничивает его функциональность по сравнению с Kubernetes;

– возможности автоматизации в Docker Swarm не такие широкие, как в Kubernetes.

Docker Swarm — подходящий вариант для развертывания небольших и средних систем, поскольку его легче настроить и администрировать. Кроме того, он проще в понимании и использует меньше ресурсов, чем Kubernetes. Kubernetes, с другой стороны, более сложен для понимания и обучения, но предлагает большую степень гибкости и масштабируемости, что делает его подходящим для больших систем. У Kubernetes более многочисленное и активное сообщество, в отличии от конкурентов, что приводит к разнообразию доступных ресурсов и интеграций сторонних продуктов.

Таким образом, для проектируемой СОД выбирается технология оркестрации Kubernetes благодаря её гибкости, повышенной безопасности и ориентированности под микросервисы.

Построение уточненной структуры обработки данных

Для проектирования оптимальной СОД предложена структурная схема уточненной модели системы обработки данных, представленная на рисунке 5.

Уточненная модель СОД представляет собой микросервисную структуру взаимосвязанных между собой контейнеров Docker. Контейнеры сгруппированы между собой в отдельные поды (в зависимости от назначения) и представлены в виде двух обособленных рабочих узлов, используя возможности оркестрации Kubernetes. Каждый из контейнеров в данном случае представляет собой отдельный микросервис — так как выполняет одну конкретную задачу и имеет простые каналы обмена данными.

Мастер-узел Kubernetes служит для администрирования и управления внешним и внутренним рабочими узлами. По умолчанию он настроен для автоматической работы и доступ к нему предоставляется с помощью «Контейнера главного устройства», расположенного во внутреннем рабочем узле.

Внешний рабочий узел представляет собой 3 взаимодействующих контейнера Docker, расположенных в поде 1, для авторизации и настройки СОД: «Контейнер UI», «Контейнер аутентификации и авторизации» и «Контейнер внешнее управление». «Контейнер UI» является микросервисом, служащим для отображения пользовательского интерфейса авторизации и настроек обработки СОД. «Контейнер аутентификации и авторизации» позволяет авторизироваться в системе для управления и настройки системы. «Контейнер внешнего управления» предназначен для корректного взаимодействия пользователя и ядра СОД. Все данные, используемые в этом узле, хранятся во внешней логической схеме БД 1, которая представляет собой одну из схем единой внешней системы управления базами данных (СУБД), что отвечает особенностям построения системы с использованием микросервисов.

В данной уточненной структурной схеме все отдельные логические схемы БД взаимодействуют с подами, так как иная реализация (взаимодействие непосредственно контейнеров с БД) значительно сложнее (в этом случае нужно разрабатывать отдельное ПО, в отличие от поддержки взаимодействия подов с контейнерами и БД в Kubernetes по умолчанию) и выходит за рамки уточненной модели СОД. При построении системы используется внешняя СУБД (все логические схемы БД расположены вне кластера Kubernetes) ввиду многочисленных недостатков для

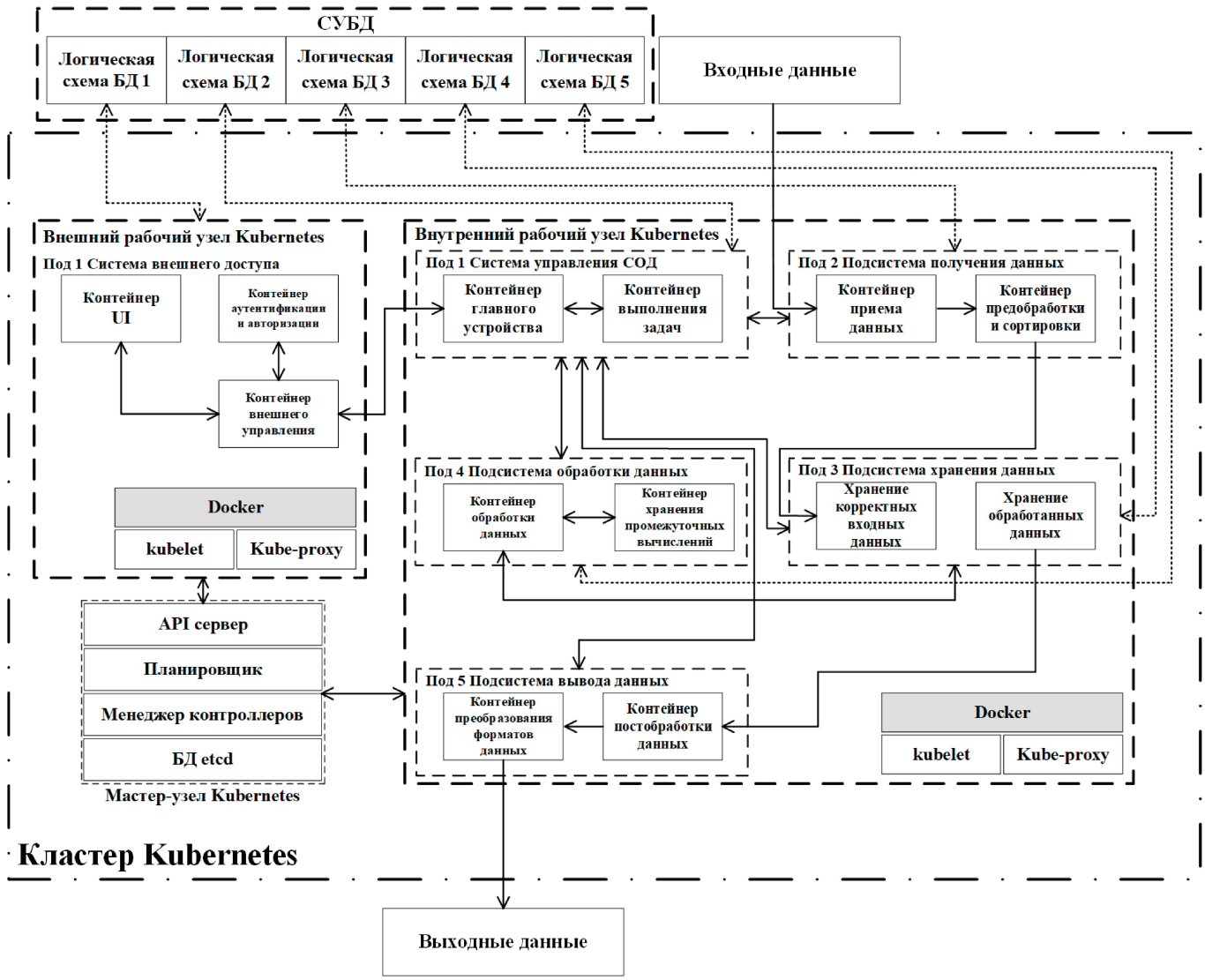

Рис. 5. **Структурная схема уточненной модели СОД**

варианта внутри кластера Kubernetes, основным из которых является значительное увеличение сложности реализации комплексной системы оркестрации [13].

Внутренний рабочий узел предназначен для обработки поступающих данных и состоит из следующих подов:

1. Система управления СОД. В этом поде располагаются «Контейнер главное устройство» и «Контейнер выполнение задач». «Контейнер главное устройство» позволяет производить администрирование и управление элементами СОД, а «Контейнер выполнения задач» играет роль планировщика и менеджера задач.

2. Подсистема получения данных служит для первоначальной обработки входящих данных и состоит из «Контейнера приема данных» и «Контейнера предобработки и сортировки». «Контейнер приема данных» отвечает за корректную передачу данных в «Контейнер предобработки и сортировки», где происходит верификация входящих данных. В процессе этой операции данные временно записываются и хранятся в логической схеме БД 3; после завершения этапа предобработки и сортировки — неверные данные удаляются.

3. Подсистема хранения данных. Этот под представляет собой совокупность контейнеров «Хранение корректных исходных данных» и «Хранение обработанных данных». В контейнер «Хранение корректных исходных данных» данные поступают из «Контейнера предобработки и сортировки», которые дополнительно проверяются и записываются в постоянное хранилище логической схемы БД 4. Контейнер «Хранение обработанных данных» служит для хранения данных после их обработки из подсистемы обработки данных.

4. Подсистема обработки данных состоит из «Контейнера обработки данных» и «Контейнера хранения промежуточных вычислений», взаимодействующих с логической схемой БД 5. В «Контейнере обработки данных» происходит операция преобразования в требуемые выходные данные, по завершению которой обработанные данные поступают в подсистему хранения данных.

5. Подсистема вывода данных содержит «Контейнер постобработки данных» и «Контейнер преобразования форматов данных». «Контейнер постобработки данных» служит для финального преобразования данных в унифицированный вид. «Контейнер преобразование форматов данных»

приводит выходные данные к требуемым пользователю/принимающей системе формату и передает их на выход.

Одним из основных преимуществ использования контейнеров Docker и средства оркестрации Kubernetes в микросервисной архитектуре разрабатываемой СОД является то, что Kubernetes в данном случае полностью отвечает за балансировку нагрузки — в любой момент времени при увеличении нагрузки на какой-либо из контейнеров возможно автоматическое создание его реплик, что позволяет в полной мере раскрыть достоинства МА.

Выводы

Предложенная уточненная модель СОД может быть полностью имплементирована в среде AWS, так как все ее основные технологии доступны в этой среде. Спроектированная СОД может являться базисом для построения определенной системы обработки данных с заданными параметрами и, в этом случае, существует возможность простой адаптации посредством организации отдельных БД для каждого из микросервисов, что значительно улучшает масштабируемость всей системы в целом. В случае выбора сложного и комплексного варианта обработки данных возможно выделение подсистемы обработки данных в отдельный рабочий узел.

Особенностью спроектированной обобщенной системы обработки данных является её гибкость, что позволяет использовать её в качестве базовой структуры в зависимости от необходимых параметров и требований к системе.

Литература:

1. Селезнёв, А. И. Обобщенная модель построения системы обработки данных / А. И. Селезнёв, И. Л. Селезнёв.— Текст: непосредственный // Молодой учёный.— 2024.— № 49.— С. 22–24.
2. Селезнёв, А. И. Контейнеризация в системах обработки данных / А. И. Селезнёв, И. Л. Селезнёв.— Текст: непосредственный // Молодой учёный.— 2023.— № 43.— С. 7–11.
3. Селезнёв, А. И. Актуальность применения микросервисной архитектуры в системах обработки данных / А. И. Селезнёв, И. Л. Селезнёв.— Текст: непосредственный // Молодой учёный.— 2023.— № 48.— С. 22–32.
4. Контейнеры или виртуальные машины — что выбрать? // Xelent: [сайт].— URL: https://www.xelent.ru/blog/konteynery-ili-virtualnye-mashiny-chto-vybrat/ (дата обращения: 29.08.2024)
5. Monolithic Approach vs. Microservices Approach: Which is Right for Your Application? // Linkedin: [сайт].— URL: https://www.linkedin.com/pulse/monolithic-approach-vs-microservices-which-right-your-majid-sheikh/ (дата обращения: 29.08.2024)
6. LXC vs Docker: Which Container Platform Is Right for You? // Earthly: [сайт].— URL: https://earthly.dev/blog/lxc-vs-docker/ (дата обращения: 29.08.2024)
7. Docker Swarm против Kubernetes: обзор и сравнение // Timeweb.cloud: [сайт].— URL: https://timeweb.cloud/blog/docker-swarm-i-kubernetes-obzor-i-sravnenie/ (дата обращения: 29.08.2024)
8. Docker Swarm vs Kubernetes: how to choose a container orchestration tool // Circleci: [сайт].— URL: https://circleci.com/blog/docker-swarm-vs-kubernetes/ (дата обращения: 29.08.2024)
9. Docker Swarm vs Kubernetes: A Practical Comparison // Betterstack: [сайт].— URL: https://betterstack.com/community/guides/scaling-docker/docker-swarm-kubernetes/ (дата обращения: 29.08.2024)
10. Docker Swarm: A Complete Guide for Beginners // k21academy: [сайт].— URL: https://k21academy.com/docker-kubernetes/docker-swarm/ (дата обращения: 29.08.2024)
11. RAFT Protocol // Medium: [сайт].— URL: https://medium.com/@swayamraina/raft-protocol-f710da8621a7/ (дата обращения: 29.08.2024)
12. Docker Swarm vs Kubernetes: выбираем фаворита // Corpsoft24: [сайт].— URL: https://www.corpsoft24.ru/about/blog/docker-swarm-vs-kubernetes-vybiraem-favorita/ (дата обращения: 29.08.2024)
13. Database in Kubernetes: Is that a good idea? // Medium: [сайт].— URL: https://medium.com/@fengruohang/database-in-kubernetes-is-that-a-good-idea-daf5775b5c1f/ (дата обращения: 29.08.2024)