Министерство науки и высшего образования
Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования «Рыбинский государственный
авиационный технический университет имени П. А. Соловьева»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ УПРАВЛЕНИЯ

Кафедра вычислительных систем

Курсовая работа
по дисциплине:
«Вычислительные системы»
на тему:

«Основы работы Docker и Kubernetes»

Выполнил: студент группы ИВМ-24 Морозов А. А.
Руководитель: кандидат технических наук, доцент Павлов Р. В.

Рыбинск 2024

Содержание

Введение

Современные вычислительные системы становятся все более сложными. Рост объемов данных, необходимость быстрого масштабирования и требования к надежности и отказоустойчивости вынуждают разработчиков и администраторов искать новые подходы к разработке и эксплуатации программного обеспечения. Одним из таких подходов является контейнеризация, которая позволяет создавать изолированные среды для приложений, что облегчает их переносимость, развертывание и управление.

Контейнеризация приобрела популярность благодаря своей легковесности и эффективности по сравнению с виртуализацией. Контейнеры используют общий системный ресурс (операционную систему), что делает их более производительными, чем виртуальные машины. Одним из самых известных инструментов для контейнеризации является *Docker*, который упрощает создание, упаковку и развертывание контейнеров.

Однако с ростом числа контейнеров в инфраструктуре встала проблема их управления. На смену простым сценариям использования *Docker* пришла потребность в масштабных решениях для оркестрации контейнеров, таких как *Kubernetes*. Эта система предоставляет возможности для автоматизации задач масштабирования, распределения нагрузки, управления состоянием приложений и их мониторинга.

Вместе *Docker* и *Kubernetes* формируют мощный инструментальный набор, который меняет подходы к разработке и эксплуатации вычислительных систем. Их использование позволяет компаниям ускорить внедрение новых решений, минимизировать время простоя, а также оптимизировать расходы на ресурсы. В данном реферате рассматриваются ключевые особенности и функции *Docker* и *Kubernetes*, их взаимодействие и преимущества, которые они предоставляют разработчикам и системным администраторам.

1 Эволюция работы веб-приложений

Развитие веб-приложений и вычислительных систем началось задолго до появления современных технологий виртуализации и контейнеризации, которые сегодня воспринимаются как неотъемлемая часть инфраструктуры IT. Ранние этапы этого пути были связаны с совсем другим подходом к разработке и эксплуатации программного обеспечения. На заре вычислительных технологий, в середине XX века, программное обеспечение создавалось исключительно под конкретные аппаратные платформы. Каждое приложение представляло собой монолитную систему, жестко связанную с ресурсами физического сервера, который играл центральную роль в обеспечении работы программ.

1.1 Монолитная архитектура

Физические серверы, могли обслуживать лишь одно приложение или один набор задач. Это было связано с архитектурными особенностями: отсутствием многозадачности и изоляции процессов. Любая программа имела доступ к ресурсам системы напрямую, без четкого разграничения, что повышало риск конфликтов и ошибок. Все компоненты приложения – вычислительная логика, база данных и интерфейс – разрабатывались как единое целое. Такой подход, известный как монолитная архитектура, подразумевал, что приложение функционирует как неразделимая единица, где все компоненты тесно связаны и зависят друг от друга.

Даже если сервер обладал избыточными вычислительными ресурсами, они часто простаивали, так как резервировались под одно приложение. Расширение таких систем также было сложным и дорогим процессом. Для повышения производительности приходилось приобретать новые физические машины, конфигурировать их и интегрировать в существующую сеть. Это требовало значительных временных и финансовых затрат, что ограничивало масштабируемость систем.

К преимуществам монолитной архитектуры можно отнести:

- простое развертывание. Использование одного исполняемого файла или каталога упрощает развертывание;

- разработка. Приложение легче разрабатывать, когда оно создано с использованием одной базы кода;

- производительность. В централизованной базе кода и репозитории один интерфейс API часто может выполнять ту функцию, которую при работе с микросервисами выполняют многочисленные API;

- упрощенное тестирование. Монолитное приложение представляет собой единый централизованный модуль, поэтому сквозное тестирование можно проводить быстрее, чем при использовании распределенного приложения;

- удобная отладка. Весь код находится в одном месте, благодаря чему становится легче выполнять запросы и находить проблемы.

Монолитные приложения работают достаточно эффективно до тех пор, пока они не становятся слишком большими и не вызывают проблем с масштабированием. Чтобы внести небольшое изменение в одну функцию, необходимо выполнить компиляцию и тестирование всей платформы, что противоречит *agile*-подходу (короткие по времени разработки с итерационным подходом к созданию и обновлению приложения), которому отдают предпочтение современные разработчики.

К недостаткам монолитной архитектуры можно отнести следующие особенности:

- снижение скорости разработки. Большое монолитное приложение усложняет и замедляет разработку;

- масштабируемость. Невозможно масштабировать отдельные компоненты;

- надежность. Ошибка в одном модуле может повлиять на доступность всего приложения;

- препятствия для внедрения технологий. Любые изменения в инфраструктуре или языке разработки влияют на приложение целиком, что зачастую приводит к увеличению стоимости и временных затрат;

- недостаточная гибкость. Возможности монолитных приложений ограничены используемыми технологиями;

- развертывание. При внесении небольшого изменения потребуется повторное развертывание всего монолитного приложения.

На рисунке 1.1 изображено схематическое представление монолитного приложения социальной сети.

Рисунок 1.1 – Монолитное приложение

1.2 Микросервисная архитектура

Из рисунка 1.1 видно, что всё монолитное приложение можно разделить на 4 приложения, которые отвечают за свои конкретные действия. Как раз разделение монолитного приложения на несколько приложений, которые как бы

обслуживают (от английского *service* – обслуживание) и послужило созданию микросервисной архитектуры. Эти приложения могут выполняться на разных серверах и связываться по сети (рисунок 1.2), но само общее приложение социальной сети так и будет работать корректно.

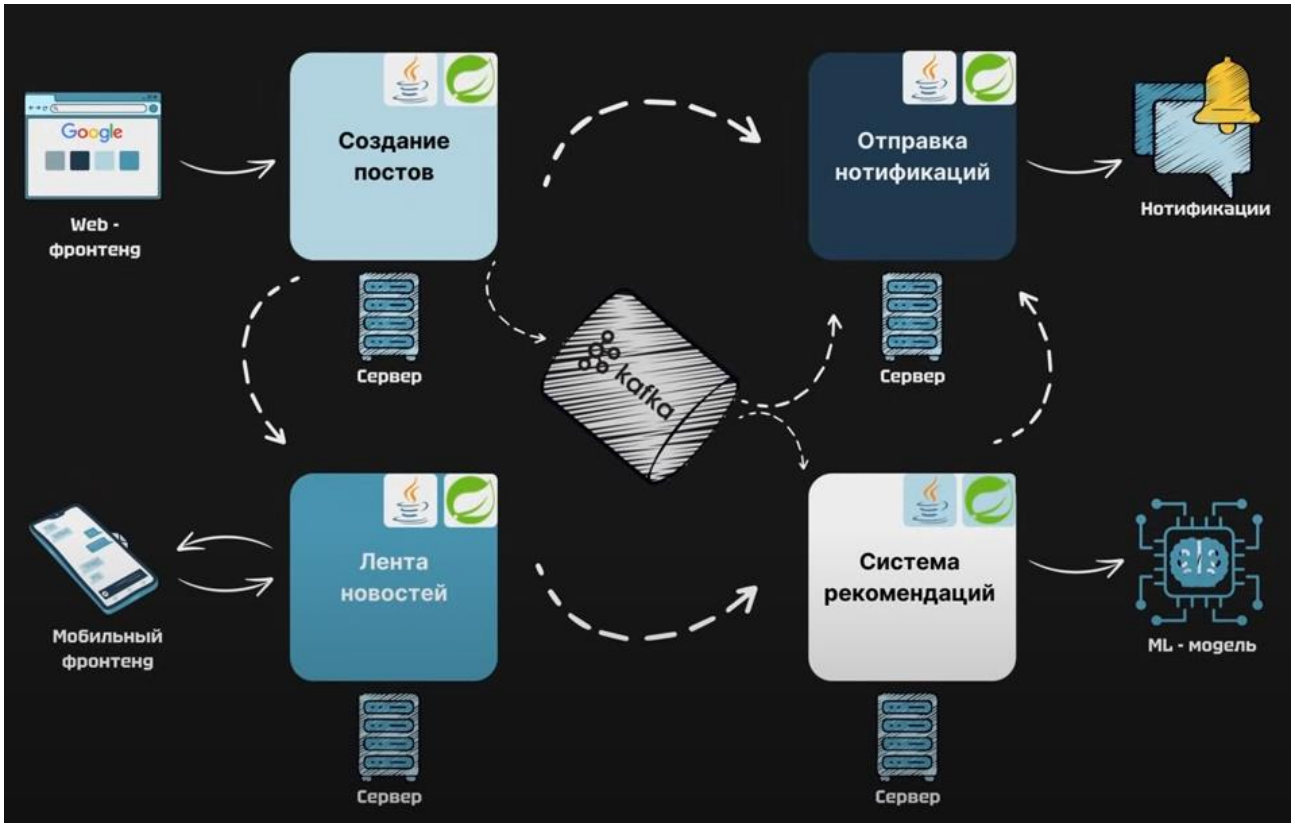

Рисунок 1.2 – Микросервисное приложение

К преимуществам микросервисов относятся:

- гибкость. Продвигайте гибкие методы работы среди небольших команд, которые регулярно выполняют развертывание;

- гибкое масштабирование. Когда микросервис достигает предельной нагрузки, можно быстро выполнить развертывание новых экземпляров данной службы в сопутствующем кластере и снизить нагрузку.

- непрерывное и независимое развертывание;

- легкость обслуживания и тестирования. Разработчик может намного проще добавлять или удалять какие-либо функции без необходимости в тестировании и отладке всех модулей;

- гибкость технологий. При использовании архитектуры микросервисов команды могут выбирать инструменты с учетом своих предпочтений.

- высокая надежность. Развертывая изменения для конкретной службы, можно не бояться, что приложение выйдет из строя целиком.

2 Виртуальная машина

С появлением микросервисной архитектуры появилась возможность использовать виртуальные машины. «Виртуальная машина (ВМ) – это виртуальный компьютер со всеми виртуальными устройствами и виртуальным жёстким диском, на который и устанавливается новая независимая операционная система (гостевая операционная система), вместе с виртуальными драйверами устройств, управлением памятью и другими компонентами.» [1].

Виртуализация предоставляет возможность создать абстракцию физического оборудования, благодаря чему на одном компьютере можно запустить множество виртуальных машин. Для приложений виртуальные машины выглядят как настоящие компьютеры, так как виртуальное оборудование отображается в системе со всеми соответствующими характеристиками. При этом каждая виртуальная машина полностью изолирована от реального оборудования, хотя может использовать ресурсы хост-компьютера, такие как дисковое пространство и периферийные устройства.

Развитие технологий виртуализации стало возможным благодаря увеличению производительности современного аппаратного обеспечения, как для серверов, так и для персональных компьютеров. Эти технологии позволяют запускать несколько операционных систем на одном физическом устройстве (хосте), что обеспечивает их независимость от конкретной аппаратной платформы. Виртуализация помогает объединить множество виртуальных машин на одном физическом сервере, что значительно экономит ресурсы.

Ключевые преимущества виртуализации включают сокращение расходов на оборудование и его обслуживание, повышение гибкости *IT*-инфраструктуры, а также упрощение процессов резервного копирования и восстановления данных. Виртуальные машины, будучи изолированными и независимыми от конкретного оборудования, могут работать на любой совместимой аппаратной платформе, что делает их универсальным решением для современных вычислительных систем.

Если раньше без виртуальных машин программист всё делал на 1 физическом компьютере, в 1 операционной системе, то с использованием микросервисной архитектуры и виртуализации программист превращался в некоего системного администратора, так как от него требовалось на 1 физический сервер установить сразу несколько гостевых систем для своих микросервисов и настроить работу их всех. Виртуализация позволяет распределять нагрузку на несколько физических серверов и производить расширение мощностей горизонтально, но заставляет программиста снова администрировать новые гостевые системы (рисунок 2.1).

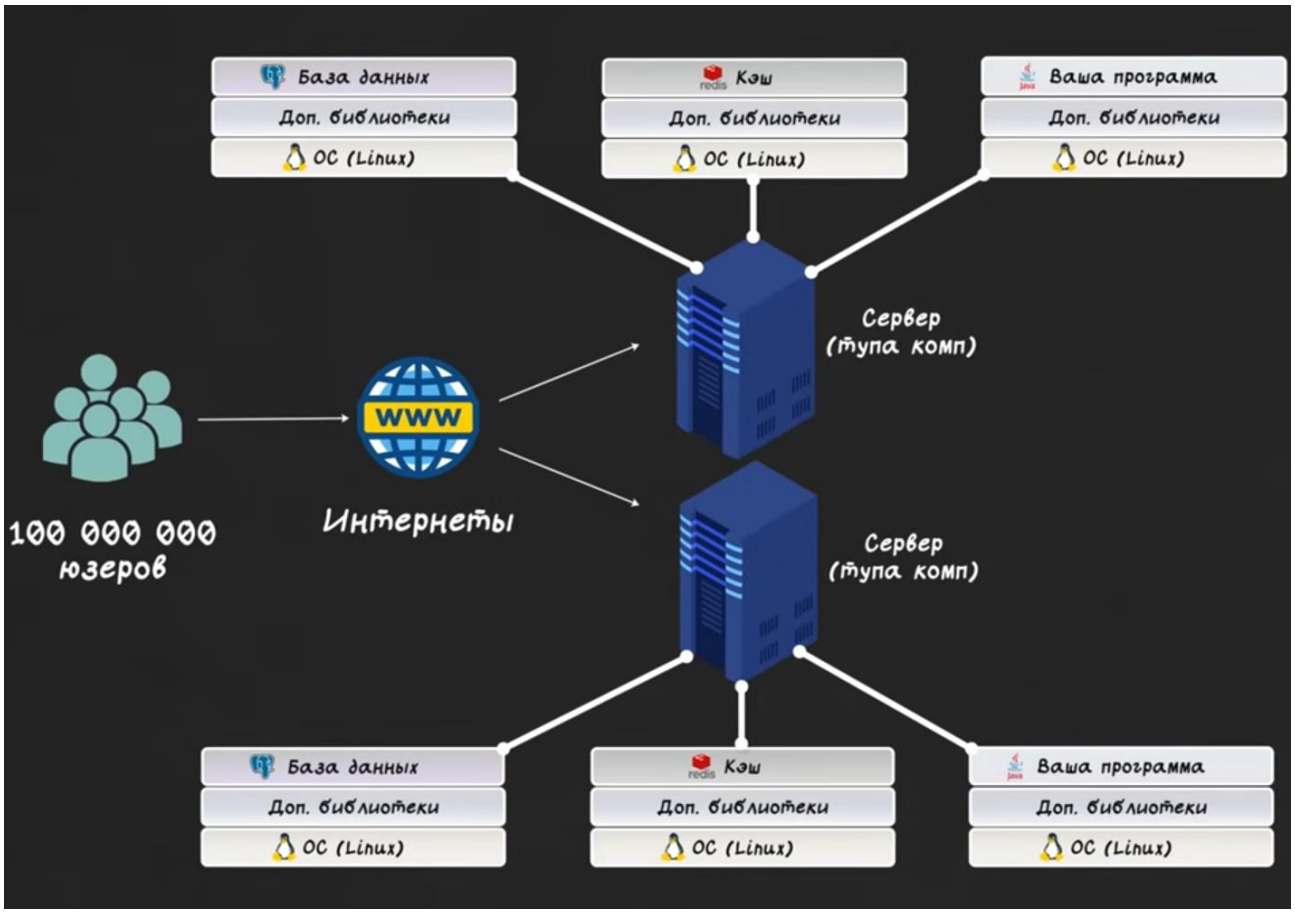

Рисунок 2.1 – Использование виртуальных машин

3 *Docker*

Из-за большой нагрузки на программиста из-за установки и настройки виртуальных машин стали использовать *Docker* [2].

Docker – это платформа с открытым исходным кодом, которая используется для разработки, упаковки и развертывания приложений в контейнерах. Контейнеры предоставляют изолированную среду выполнения, которая содержит все необходимые для работы приложения компоненты: код, зависимости, библиотеки и конфигурации. Эта технология позволяет запускать приложения одинаково на разных системах, независимо от их аппаратной или программной среды.

3.1 *Docker Image*

Для начала работы необходимо создать так называемый *Docker Image*. *Docker Image* (образ *Docker*) – это неизменяемый файл, содержащий всё необходимое для запуска приложения внутри контейнера. Образ можно рассматривать как шаблон или основу, из которой создаются контейнеры. Он включает операционную систему, приложения, библиотеки, зависимости и любые конфигурации, необходимые для корректной работы программного обеспечения (рисунок 3.1). В качестве операционной системы можно использовать *alpine*, объём которой составляет лишь 5 МБ [3]. После создания образа его можно загрузить в репозиторий для хранения и обмена. Наиболее популярным является *Docker Hub* – центральный репозиторий *Docker*, где можно найти образы для различных операционных систем, баз данных, веб-серверов и других инструментов. Репозитории могут быть как и публичными, так и приватными.

Рисунок 3.1 – Образ *Docker*

Созданный образ необходимо перенести на сервер и запустить через приложение *Docker*. После запуска образа программа, которая в нём хранилась, успешно запустится.

3.2 *Dockerfile*

Для создания *Docker Image* программисту необходимо написать так называемый *Dockerfile*. В нём используется концепция слоёв, каждый из которых начинается со специфичных инструкций (*FROM*, *RUN*, *COPY*, *ADD*). Каждый новый слой содержит только новую информацию и *Docker* кэширует каждый слой, что позволяет экономить место, время для сборки образов и развёртывание контейнеров.

Dockerfile – это текстовый файл, содержащий последовательность инструкций, которые *Docker* использует для создания образа. Он позволяет автоматизировать процесс сборки *Docker Image*, задавая необходимые

параметры, такие как базовый образ, команды для установки программ, копирование файлов и настройка окружения.

Чтобы разработчику не приходилось постоянно вручную искать и скачивать на своё устройство нужные утилиты для создания *Docker Image* он может с помощью команды *FROM* указать название репозитория в *Docker Hub* и сам *Docker* его скачает.

3.3 *Docker Container*

Docker Container (контейнер *Docker*) – это исполняемая версия *Docker Image*. Контейнер представляет собой изолированную среду, в которой запускается приложение, полностью отделённое от системы хоста и других контейнеров. Контейнеры используют общие ресурсы операционной системы, что делает их лёгкими и производительными по сравнению с виртуальными машинами.

Контейнер – это работающий экземпляр образа *Docker*, который включает в себя всё необходимое для выполнения приложения: код, зависимости, библиотеки и системные настройки. Однако контейнеры отличаются от виртуальных машин тем, что не включают отдельную операционную систему. Вместо этого они используют ядро хостовой операционной системы, минимизируя накладные расходы. Если ядро системы на сервере и ядро в контейнере будут различаться, то *Docker* будет использовать технологию *Windows Subsystem for Linux* (*WSL*) для успешной работы контейнера. Данная технология интерпретирует системные вызовы *Linux* и преобразует их в эквиваленты *Windows*.

Контейнеры отличаются от виртуальных машин в меньшем объёме, моментальный запуск и сниженным потреблением ресурсов.

Именно благодаря контейнерам *Docker* и стал таким популярным.

4 *Kubernetes*

От современных веб-сервисов пользователи ожидают, что приложения будут доступны 24 часа в сутки, 7 дней в неделю, а разработчики развёртывать новые версии приложений по нескольку раз в день. Контейнеризация направлена на достижение этой цели, поскольку позволяет выпускать и обновлять приложения без простоев.

Kubernetes гарантирует, что ваши контейнеризованные приложения будет запущены где угодно и когда угодно, вместе со всеми необходимыми для их работы ресурсами и инструментами. *Kubernetes* это готовая к промышленному использованию платформа с открытым исходным кодом, разработанная на основе накопленного опыта *Google* по оркестровке контейнеров и вобравшая в себя лучшие идеи от сообщества [4].

Kubernetes состоит из множества частей [2]:

- *pods* (поды);
- деплоймент (развёртывание);
- узел (нода);
- кластер;
- постоянный том.

4.1 *Pods*

После создания контейнера с помощью *Docker* его необходимо обернуть в структуру более высокого уровня, которая называется подом. Все контейнеры в одном поде будут использовать одни и те же ресурсы и локальную сеть (рисунок 4.1). Контейнеры могут легко взаимодействовать с другими контейнерами в одном модуле, как если бы они находились на одном компьютере, сохраняя при этом определенную степень изоляции от других.

Поды используются в качестве единицы репликации в *Kubernetes*. Если ваше приложение становится слишком популярным и один экземпляр модуля не может нести нагрузку, *Kubernetes* можно настроить для развертывания новых

реплик вашего модуля в кластере по мере необходимости. Даже когда нагрузка невелика, стандартно иметь несколько копий модуля, работающих в любое время в производственной системе, чтобы обеспечить балансировку нагрузки и устойчивость к сбоям.

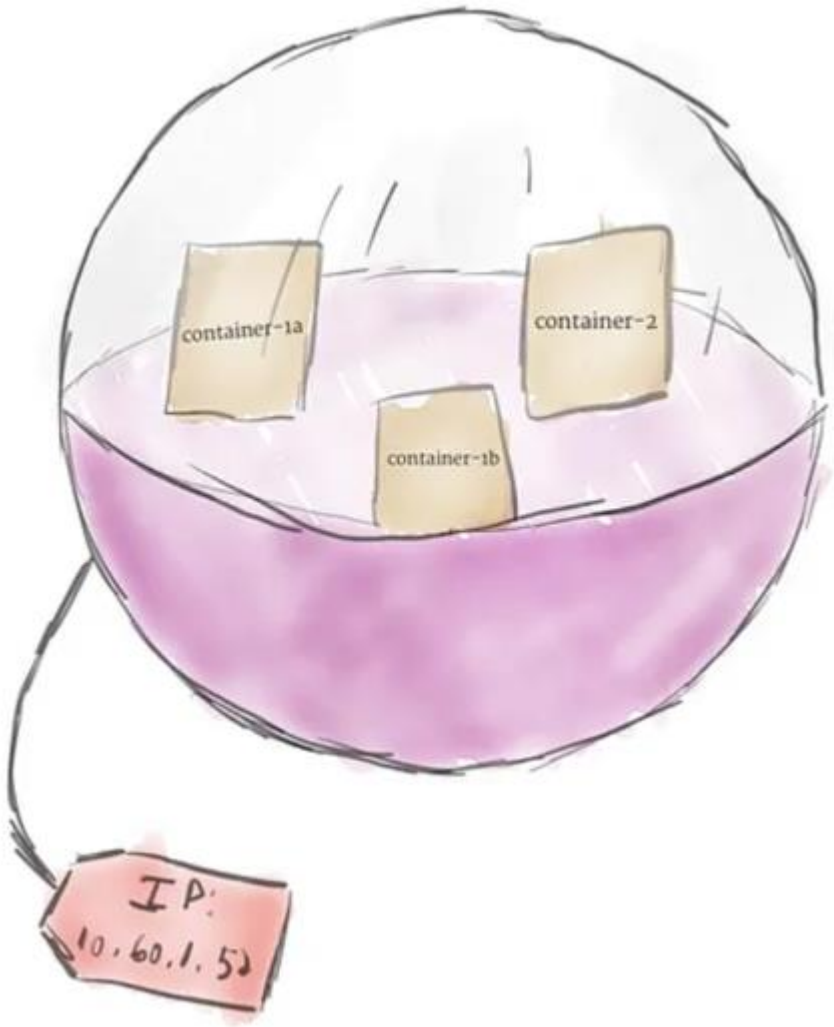

Рисунок 4.1 – Визуализация пода

4.2 Развёртывание

Хотя поды являются базовой единицей вычислений в *Kubernetes*, они обычно не запускаются напрямую в кластере. Вместо этого модули обычно управляются еще одним уровнем абстракции: развертыванием (рисунок 4.2).

Основная цель развертывания – объявить, сколько реплик пода должно работать одновременно. Когда развертывание добавляется в кластер, оно автоматически запускает запрошенное количество модулей, а затем отслеживает их. Если под умирает, развертывание автоматически воссоздает его.

Используя развертывание, программисту не нужно иметь дело с подами вручную. Можно просто объявить желаемое состояние системы, и оно будет управляться автоматически.

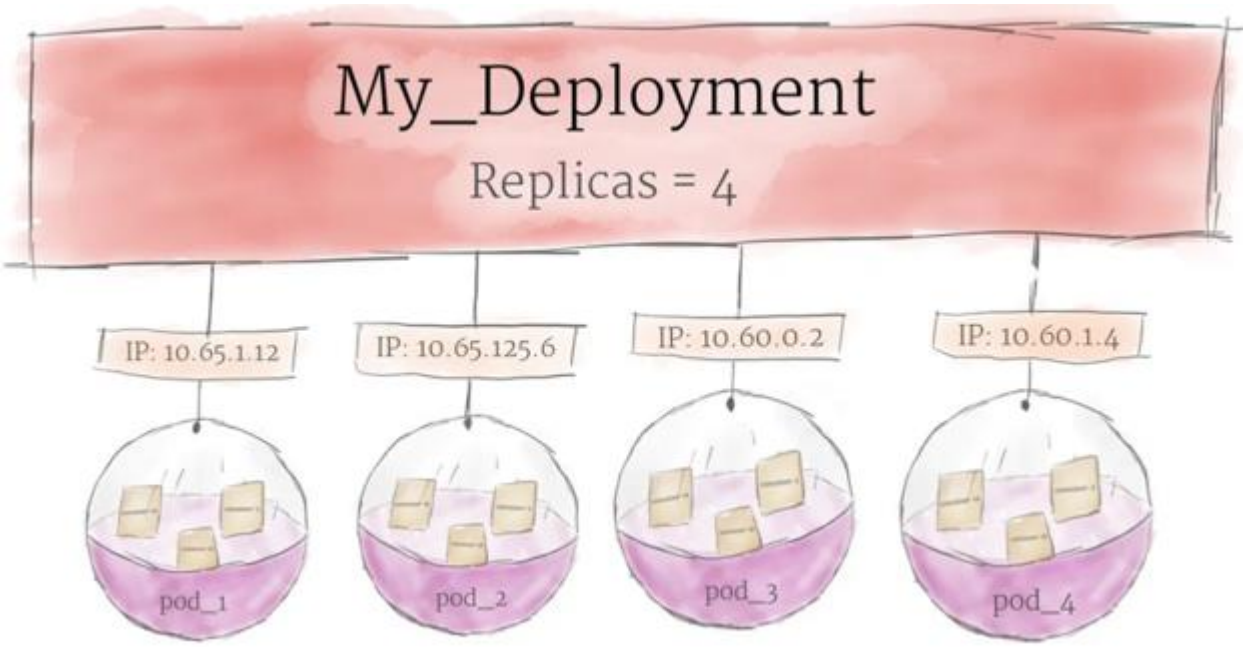

Рисунок 4.2 – Визуализация развёртывания

4.3 Узел

Узел – это наименьшая единица вычислительного оборудования в *Kubernetes*. Это представление одной машины в кластере. В большинстве производственных систем узел, скорее всего, будет либо физической машиной в центре обработки данных, либо виртуальной машиной, размещенной у облачного провайдера, такого как *Google Cloud Platform*. Теоретически можно сделать узел практически из чего угодно.

Представление о машине как об «узле» позволяет добавить слой абстракции. Теперь вместо того, чтобы беспокоиться об уникальных характеристиках каждой отдельной машины, можно просто рассматривать каждую машину как набор ресурсов ЦП и ОЗУ, которые можно использовать. Таким образом, любая машина может заменить любую другую машину в кластере *Kubernetes*.

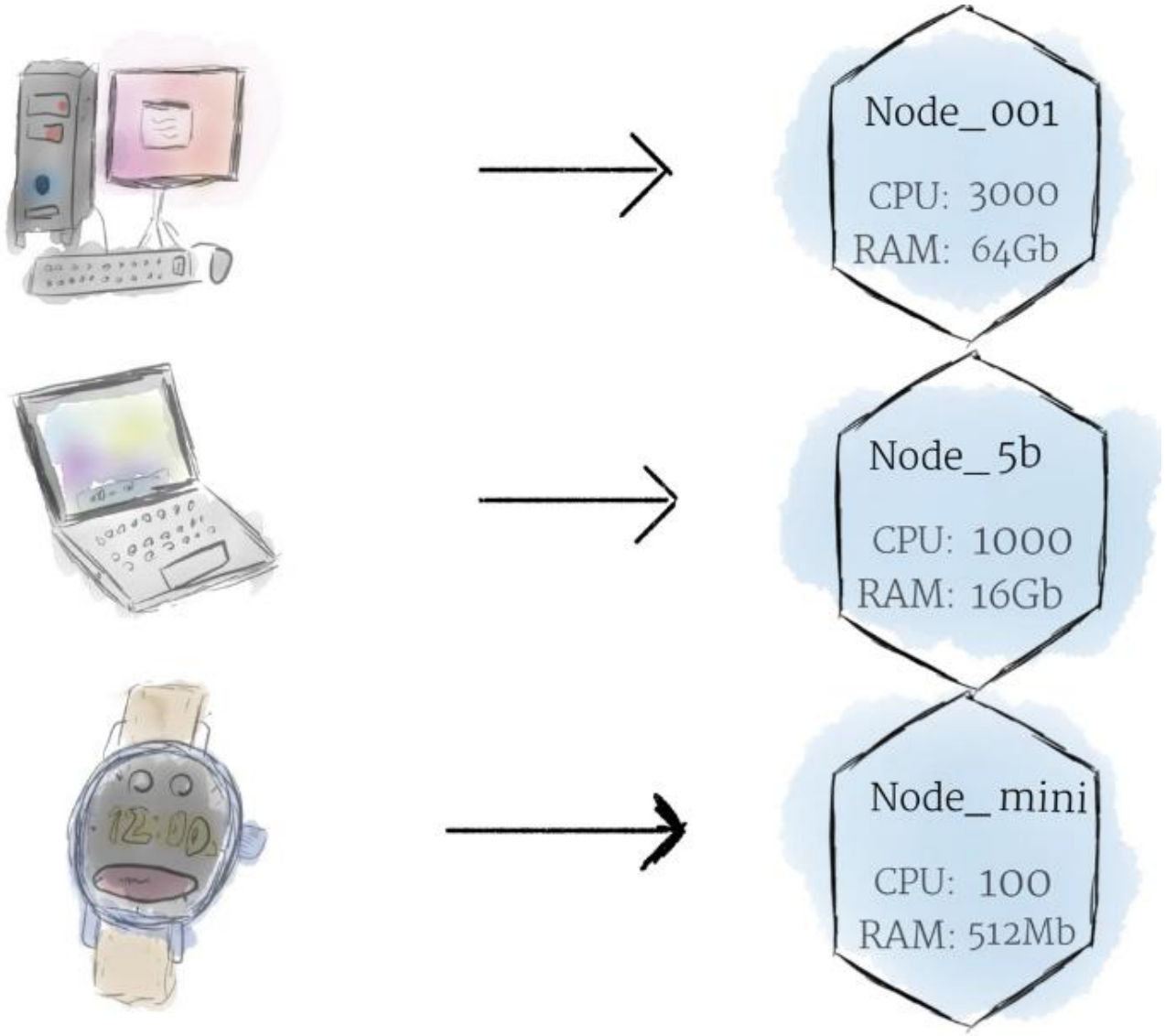

Рисунок 4.3 – Визуализация узла

4.4 Кластер

В *Kubernetes* узлы объединяют свои ресурсы, чтобы сформировать более мощную машину (рисунок 4.4). Когда разработчик развертывает программы в кластере, он интеллектуально распределяет работу по отдельным узлам за вас. Если какие-либо узлы будут добавлены или удалены, кластер будет переключаться между работой по мере необходимости. Для программы или программиста не должно иметь значения, на каких машинах фактически выполняется код.

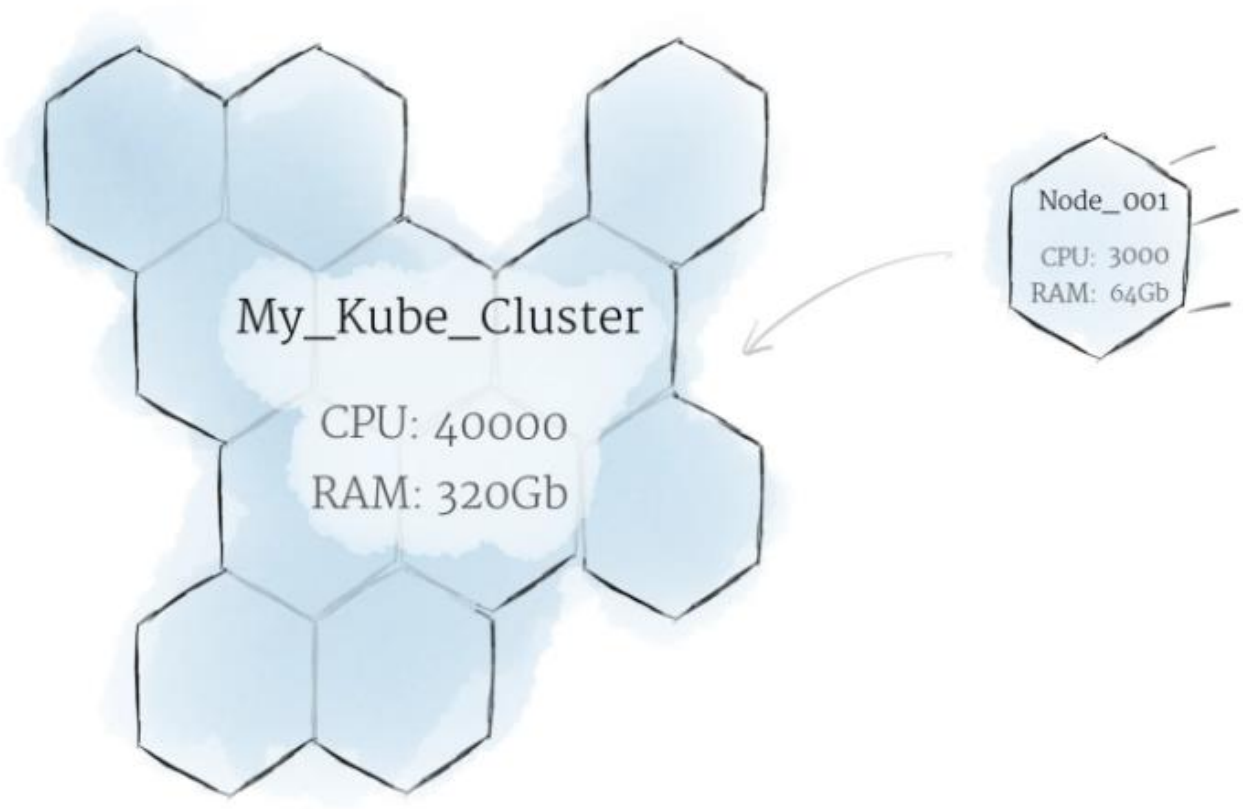

Рисунок 4.4 – Визуализация кластера

4.5 Постоянный том

Поскольку программы, работающие в кластере, не гарантируют работу на определенном узле, данные не могут быть сохранены в любом произвольном месте в файловой системе. Если программа попытается сохранить данные в файл на потом, но затем будет перемещена на новый узел, файл больше не будет находиться там, где программа ожидает его видеть. По этой причине традиционное локальное хранилище, связанное с каждым узлом, рассматривается как временный кэш для хранения программ, но нельзя ожидать, что любые данные, сохраненные локально, сохранятся.

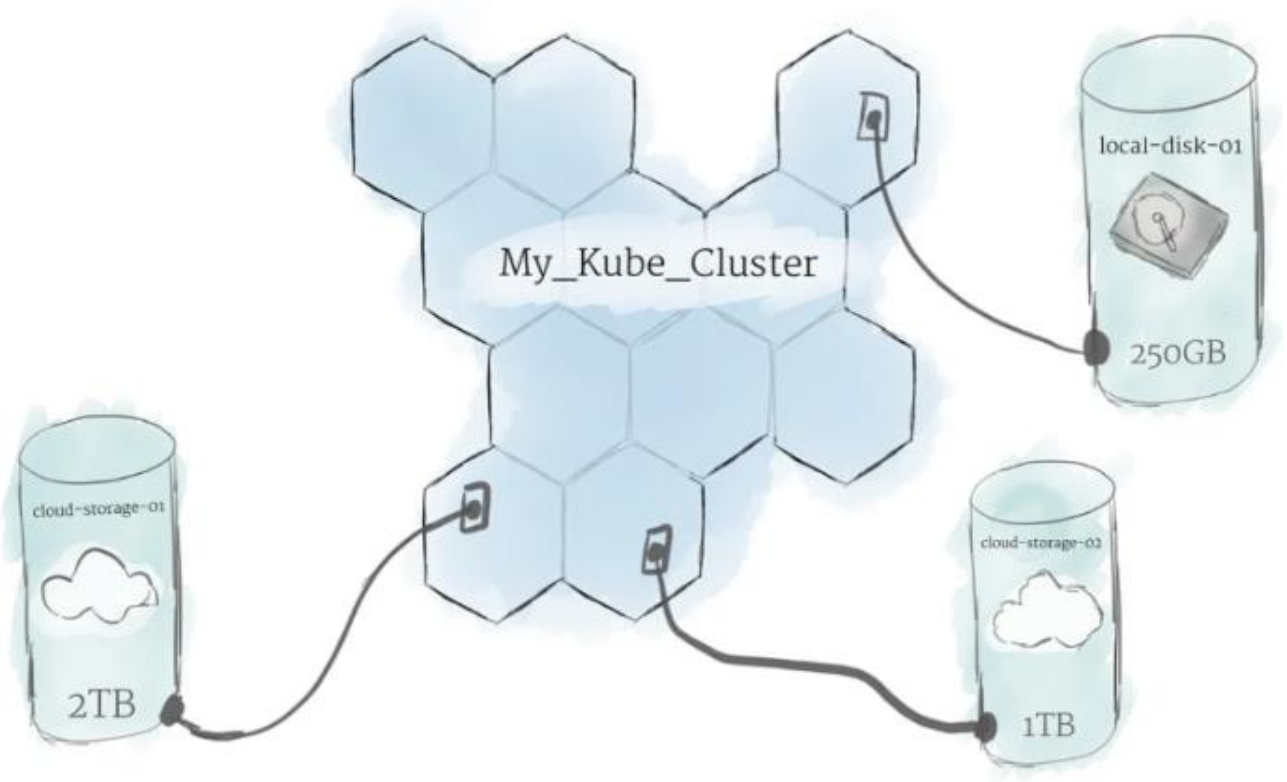

Рисунок 4.4 – Визуализация постоянных томов

Для постоянного хранения данных *Kubernetes* использует постоянные тома. В то время как ресурсы ЦП и ОЗУ всех узлов эффективно объединены и

управляются кластером, постоянное хранилище файлов – нет. Вместо этого локальные или облачные диски могут быть подключены к кластеру как постоянный том. Это можно рассматривать как подключение внешнего жесткого диска к кластеру. Постоянные тома предоставляют файловую систему, которую можно подключить к кластеру без привязки к какому-либо конкретному узлу.

Заключение

Технологии Docker и Kubernetes значительно изменили подход к разработке, развертыванию и управлению веб-приложениями. Docker, с его концепцией контейнеризации, предоставляет гибкость и изоляцию, позволяя разработчикам легко развертывать и масштабировать свои приложения. Dockerfile и Docker Image играют ключевую роль в создании и поддержании стабильных и повторяемых сред для разработки и тестирования, обеспечивая чёткую среду выполнения для контейнеров.

Kubernetes, в свою очередь, предлагает мощное решение для автоматизации управления контейнерными приложениями. Его архитектура, включающая различные компоненты для контроля состояния, сетевого взаимодействия и безопасности, позволяет разработчикам сосредоточиться на бизнес-логике своих приложений, а не на инфраструктуре. Единицы репликации, управляемые через развёртывание, обеспечивают автоматическое масштабирование и стабильность подов, а также высокую доступность приложений в кластере.

Вместе *Docker* и *Kubernetes* создают мощный инструмент для современных распределённых систем, позволяя разработчикам создавать, развертывать и управлять приложениями в облачных и локальных средах. Их использование способствует повышению производительности, улучшению контроля над ресурсами и упрощению процессов интегрирования и развёртывания обновлений, что делает их незаменимыми в современной веб-разработке. Благодаря этим технологиям разработчики могут сосредоточиться на написании кода, зная, что их приложения будут автоматически масштабироваться, изолироваться и управляться с высокой степенью надежности и безопасности.

Список использованных источников

1. Сергеева О. О., Белозерова А. Р. *Kubernetes* с *Docker* на локальной машине – Научная статья – Димитровград : МИФИ, 2020 с. 44-53.

2. *Docker Docs* – сайт с документацией [Электронный ресурс] : Электронные текстовые данные. *URL* : https://docs.docker.com/get-started/ (дата обращения: 07.12.2024).

3. Alpine – a minimal Docker image based on Alpine Linux with a complete package index and only 5 MB in size! [Электронный ресурс]. *URL* : *https*://*hub.docker.com*/*_*/*alpine* (дата обращения: 07.12.2024).

4. *Kubernetes* – сайт с документацией [Электронный ресурс] : Электронные текстовые данные. *URL* : *https*://*kubernetes.io*/*docs*/*concepts*/*overview*/.