DOI: 10.58168/PPSPM2025_416-423
УДК 004.9

МИКРО СЕРВИСНАЯ АРХИТЕКТУРА КАК ИНСТРУМЕНТ ПОВЫШЕНИЯ ОТКАЗОУСТОЙЧИВОСТИ ИНФОРМАЦИОННЫХ СИСТЕМ

А.И. Литвинов, О.В. Оксюта

ФГБОУ ВО «Воронежский государственный лесотехнический университет имени Г.Ф. Морозова»

В работе рассматривается микросервисная архитектура как альтернатива монолитной для повышения надежности информационных систем. На примере системы, состоящей из чат-бота и RAG-сервиса, проведено сравнение двух подходов. Практическая реализация на Docker Compose демонстрирует принцип изящной деградации при отказе одного из сервисов. Рассмотрены преимущества оркестратора Kubernetes для автоматизации восстановления и масштабирования системы.

Ключевые слова: микросервисная архитектура, монолитная архитектура, отказоустойчивость, Docker, Kubernetes, RAG, чат-бот.

MICRO-SERVICE ARCHITECTURE AS A TOOL FOR INCREASING INFORMATION SYSTEM FAULT TOLERANCE

A.I. Litvinov, O.V. Oksiuta

Voronezh State University of Forestry and Technologies named after G.F. Morozov

The paper considers microservice architecture as an alternative to monolithic architecture for improving the reliability of information systems. Using the example of a system consisting of a chatbot and a RAG service, a comparison of the two approaches is carried out. A practical implementation using Docker Compose demonstrates the principle of graceful degradation when one of the services fails. The advantages of the Kubernetes orchestrator for automating recovery and system scaling.

Key words: microservice architecture, monolithic architecture, fault tolerance, Docker, Kubernetes, RAG, chatbot.

В текущих условиях повсеместной цифровизации и для бизнеса, и для государства важно обеспечить бесперебойную работу своих цифровых услуг. Однако стабильности мешает постоянное развитие и усложнение электронных сервисов, до сих пор широко используемая монолитная структура, где все компоненты тесно связаны демонстрирует низкий уровень отказоустойчивости. В такой модели отказ любого, даже второстепенного модуля, ведет к каскадному падению всех остальных, даже стабильных компонентов.

В качестве альтернативы выступает микросервисная архитектура. Данный подход предполагает разбитие монолитной системы на более мелкие, специализированные и относительно изолированные фрагменты. Каждый сервис инкапсулирует определенную бизнес-функцию и взаимодействует с другими через четко определенные API, благодаря чему отказ работы одной из функций не ломает систему целиком, а позволяет быстро выявить проблему и исправить практически незаметно для конечного пользователя.

Целью данной работы является анализ и практическая демонстрация эффективности микросервисной архитектуры как инструмента повышения отказоустойчивости информационных систем на примере системы, состоящей из чат-бота и RAG-сервиса (Retrieval-Augmented Generation).

Для наглядной демонстрации преимуществ микросервисной архитектуры с точки зрения надежности, целесообразно начать с рассмотрения традиционного подхода — монолитной архитектуры.

Монолитное приложение представляет собой единую, тесно связанную кодовая базу, в которой компоненты пользовательского интерфейса, бизнес-логика и вспомогательные модули скомпилированы и развернуты как единое целое (Рис. 1).

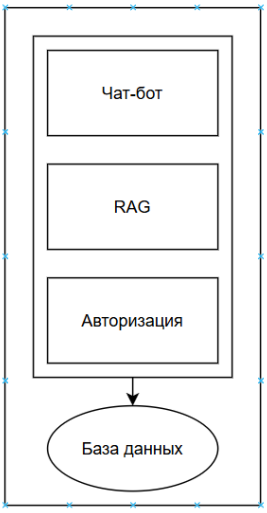

Рисунок 1 – Монолитная архитектура

Проблема архитектуры такого типа в единой точке отказа. Любая ошибка в любом, даже малозначительном модуле, способна вызвать "эффект домино" и привести к полному прекращению работы всего приложения. В такой ситуации пользователь сталкивается с ошибкой сервера или полным отсутствием ответа. Таким образом, монолитная архитектура демонстрирует низкую устойчивость к сбоям, поскольку не предусматривает механизмов изоляции отказа.

Альтернативный подход, называемый микросервисной архитектурой (Рис. 2) предлагает декомпозицию сложной системы на набор небольших, слабосвязанных и узкоспециализированных сервисов. Ключевой принцип, обеспечивающий повышенную надежность в сравнении с монолитной архитектурой — изоляция. Каждый сервис работает в собственной среде и взаимодействует с другими через четко определенные правила API.

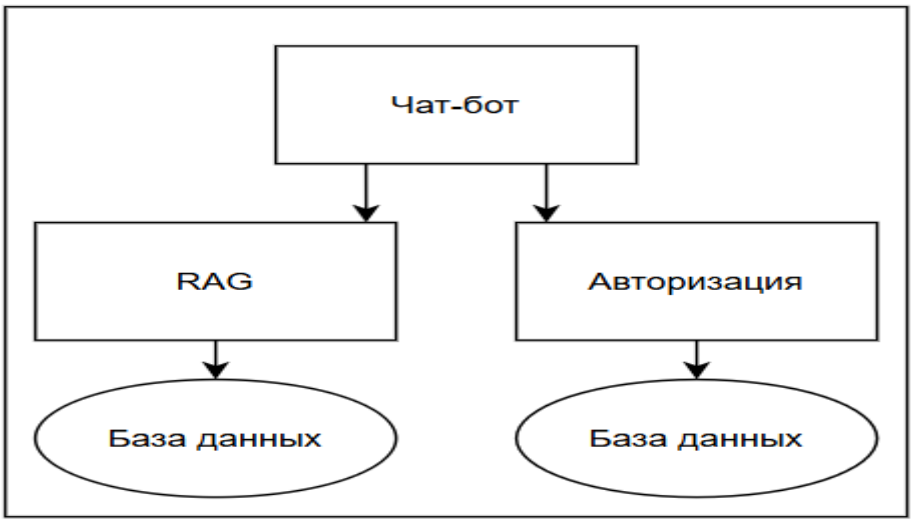

Рисунок 2 – Микросервисная архитектура

Преимущество становится очевидным в момент сбоя. Если один из сервисов (например, RAG) перестает отвечать, другие сервисы не прекращают работу. Вместо этого они фиксируют ошибку соединения и, благодаря встроенной логике обработки сбоев, реализуют принцип "изящной деградации": система теряет отдельную функцию, но сохраняет общую работоспособность.

Для проверки концепции была использована действующая система, состоящая из двух независимых компонентов:

1. Чат-бота, отвечающего за общение с пользователем.
2. RAG-сервиса, обеспечивающего семантический поиск и генерацию ответов на основе базы знаний.

Оба компонента были упакованы в изолированные Docker-контейнеры, что гарантирует возможность развертываемость в любых окружениях и отсутствие конфликтов между ними. Для организации взаимодействия и удобства развертывания сервисов используется Docker Compose. Файл docker-compose.yml описывает конфигурацию обоих сервисов и общую сеть для их взаимодействия:

```
services:
  chat:
    build:
     context: ./chat
     dockerfile: Dockerfile
     target: development
    ports:
     - "${RAG_SERVICE_URL}:8000
    environment:
     -  NODE_ENV=development
   networks:
     - app-network

  rag-service:
    build:
     context: ./rag
     dockerfile: Dockerfile
     target: development
    ports:
     - "${RAG_SERVICE_URL}:8001
    networks:
     - app-network

networks:
  app-network:
    driver: bridge
```

Взаимодействие между сервисами построено через HTTP API. Когда пользователь задает сложный вопрос, требующий поиска в базе знаний, чат-бот асинхронно вызывает RAG-сервис следующим образом:

```
async def generate_response(self, user_message: str) -> str:
```

```
if self._needs_rag(user_message):
    try:
        async with aiohttp.ClientSession() as session:
            async with session.post(
                f"{self.rag_url}/generate",
                json={"query": user_message, "model": "gpt-4"},
                timeout=client_timeout
            ) as response:
                if response.status == 200:
                    data = await response.json()
                    return data["answer"]
                else:
                    raise ServiceError("RAG service error")
    except (aiohttp.ClientError, asyncio.TimeoutError):
        logger.error("RAG service unavailable")
        return "Сервис поиска временно недоступен"

return await self._generate_basic_response(user_message)
```

Для демонстрации отказоустойчивости был протестирован сценарий выхода из строя RAG-сервиса. При остановке контейнера rag-service чат-бот продолжал работать в штатном режиме, игнорируя потерю сервиса и уведомляя пользователей о временной недоступности поиска по базе знаний. Этот эксперимент наглядно демонстрирует ключевое преимущество микросервисной архитектуры: даже при частичном отказе система сохраняет базовую функциональность и остаётся полезной для пользователя.

Хотя Docker и, в частности, его инструмент Docker Compose отлично подходят для локальной разработки и небольших систем, они имеют существенные ограничения при масштабировании. Так, управление контейнерами вручную быстро становится неудобным, а встроенных механизмов самовосстановления, например, автоматической перезагрузки упавшего сервиса или балансировки нагрузки между копиями в стандартном Docker, из "коробки" такого нет.

Здесь на помощь приходит Kubernetes (K8s) — промышленный стандарт для оркестрации контейнеров. Это платформа предназначенна для автоматизации развёртывания, масштабирования и управления жизненным циклом контейнеризованных приложений.

Kubernetes, в отличие от Docker, не просто запускает контейнеры, а следит за их состоянием и реагирует на изменение их состояния. Три основных механизма которые следят и корректируют состояние системы:

1. Liveness Probes —позволяет Kubernetes регулярно отправлять запросы к контейнерам, чтобы убедиться, что он в работе. Если проверка не проходит, K8s автоматически перезапускает его, не дожидаясь вмешательства человека.
2. Readiness Probes — Даже если контейнер запущен, он может ещё быть не готов к полноценной работе (например, загружает модель или подключается к базе). Readiness-проверки позволяют временно исключить такой экземпляр из балансировки, пока он не станет полностью функциональным.
3. ReplicaSets и Deployments — гарантия доступности. Он позволяет автоматически подменять важный сервис с помощью реплик (например, RAG-поиска), которых минимум три. Если одна из них падает — Kubernetes мгновенно запускает новую на любом доступном узле кластера.

Благодаря этим возможностям Kubernetes выводит отказоустойчивость на новый уровень: от простой изоляции сбоев (как в микросервисах на Docker) к автоматическому реагированию на внештатные ситуации. Система становится не просто устойчивой, а самовосстанавливающийся — она сама обнаруживает проблемы и принимает меры, чтобы минимизировать влияние ошибки на пользователя.

В перспективе переход от Docker Compose к Kubernetes позволяет не только повысить надёжность, но и гибко масштабировать отдельные компоненты под нагрузку и упростить обновления.

Таким образом, микросервисная архитектура существенно повышает отказоустойчивость информационных систем. На примере системы из чат-бота и RAG-сервиса продемонстрировано, что изоляция компонентов позволяет сохранить базовую функциональность даже при полном отказе одного из сервисов.

Практическая реализация на Docker показала эффективность разделения ответственности между компонентами. Для масштабируемости или более сложных приложений естественным развитием становится использование Kubernetes, который автоматизирует восстановление после сбоев и обеспечивает масштабируемость системы.

Список литературы

1. Алексеев, М.К. Опыт внедрения Kubernetes для оркестрации микросервисов / М.К. Алексеев– 2023. – С. 112-120.

2. Аншина, М. Л. Архитектура приложений и данных: учебное пособие / М. Л. Аншина. — Москва: РТУ МИРЭА, 2024. — 152 с. — ISBN 978-5-7339-2218-8. // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/421100 (дата обращения: 16.10.2025). — Режим доступа: для авториз. пользователей.

3. Годзурас, Э. Docker Compose для разработчика: руководство / Э. Годзурас ; перевод с английского А. Н. Киселева. — Москва: ДМК Пресс, 2023. — 220 с. — ISBN 978-5-93700-203-7. // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/348110 (дата обращения: 16.10.2025). — Режим доступа: для авториз. пользователей.

4. Кочер, П. С. Микросервисы и контейнеры Docker : руководство / П. С. Кочер ; перевод с английского А. Н. Киселева. — Москва: ДМК Пресс, 2019. — 240 с. — ISBN 978-5-97060-739-8. // Лань: электронно-библиотечная система. — URL: https://e.lanbook.com/book/123710 (дата обращения: 16.10.2025). — Режим доступа: для авториз. пользователей.

5. Docker Documentation: официальная документация по использованию Docker // Официальный сайт Docker Inc. – URL: https://docs.docker.com/ (дата обращения: 14.10.2025).

6. Kubernetes Documentation: руководство по развертыванию и управлению кластерами // Сайт Cloud Native Computing Foundation. – URL: https://kubernetes.io/docs/ (дата обращения: 15.10.2025).

References

1. Alekseev, M.K. Experience in implementing Kubernetes for microservices orchestration / M.K. Alekseev– 2023. – pp. 112-120.

2. Anshina, M. L. Application and Data architecture: a textbook / M. L. Anshina. — Moscow: RTU MIREA, 2024. 152 p. ISBN 978-5-7339-2218-8. — Text: electronic // Lan: electronic library system. — URL: https://e.lanbook.com/book/421100 (date of request: 16.10.2025). — Access mode: for authorization. users.

3. Godzouras, E. Docker Compose for the developer: a guide / E. Godzouras ; translated from English by A. N. Kiselyov. — Moscow: DMK Press, 2023. — 220 p.

— ISBN 978-5-93700-203-7. // Lan: electronic library system. — URL: https://e.lanbook.com/book/348110 (date of request: 16.10.2025). — Access mode: for authorization. users.

4. Kocher, P. S. Microservices and Docker containers: a guide / P. S. Kocher ; translated from English by A. N. Kiselyov. — Moscow: DMK Press, 2019. 240 p. ISBN 978-5-97060-739-8. / Lan: electronic library system. — URL: https://e.lanbook.com/book/123710 (date of request: 16.10.2025). — Access mode: for authorization. users.

5. Docker Documentation: official documentation on using Docker // Official website of Docker Inc. – URL: https://docs.docker.com / (date of request: 14.10.2025).

6. Kubernetes Documentation: A guide to cluster Deployment and management // Cloud Native Computing Foundation website. – URL: https://kubernetes.io/docs / (date of request: 10/15/2025).