МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра дискретной математики и информационных технологий

ВЫЯВЛЕНИЕ ПРЕИМУЩЕСТВ РЕАЛИЗАЦИИ НА ЯЗЫКЕ GO МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ ПЕРЕД МОНОЛИТНЫМ ПРИЛОЖЕНИЕМ

АВТОРЕФЕРАТ МАГИСТЕРСКОЙ РАБОТЫ

студента 2 курса 271 группы
направления 09.04.01 — Информатика и вычислительная техника
факультета КНиИТ
Давиденко Алексея Алексеевича

Научный руководитель
доцент, к. ф.-м. н. _____ В. А. Поздняков

Заведующий кафедрой
доцент, к. ф.-м. н. _____ Л. Б. Тяпаев

Саратов 2023

ВВЕДЕНИЕ

В настоящее время разработка программного обеспечения становится все более сложным и требовательным процессом. Один из ключевых факторов успеха в разработке - это выбор подходящей архитектуры, которая обеспечит эффективное и масштабируемое функционирование приложений. Две из наиболее распространенных архитектурных моделей, которые привлекают все больше внимания в индустрии разработки ПО, - это монолитная архитектура и микросервисная архитектура.

Монолитная архитектура представляет собой подход, при котором весь код приложения находится в одном монолитном блоке, и все функции и компоненты приложения работают в контексте этого блока. Это означает, что весь функционал приложения выполняется в едином процессе. Монолитная архитектура имеет ряд преимуществ, таких как возможность быстрого создания прототипов приложений, простота внесения функциональных изменений и удобство отслеживания всего приложения в одном месте. Однако с увеличением размеров и сложности приложения возникают проблемы с его поддержкой и масштабируемостью.

Микросервисная архитектура, основанная на сервис-ориентированной парадигме, напротив, представляет собой подход, при котором большое приложение разбивается на мелкие автономные сервисы, каждый из которых выполняет отдельную функцию и взаимодействует с другими сервисами через API. Это позволяет достичь большей гибкости и масштабируемости приложения, так как каждый сервис может быть масштабирован и развернут независимо от других сервисов. Микросервисная архитектура также обеспечивает устойчивость и отказоустойчивость, так как отказ одного сервиса не оказывает влияния на работу других сервисов. Однако она требует более сложного управления и координации между сервисами, а также усложняет тестирование и отладку из-за разнообразия технологий и возможных ошибок в каждом сервисе.

Актуальность данной работы заключается в том, что в настоящее время всё больше разработчиков при разработке программного обеспечения выбирают архитектуры, направленные на модульность и переиспользуемость компонентов, несмотря на то что монолитная архитектура всё также является одной из самых стабильных. С ростом популярности языка Golang всё больше компаний начали процесс по переходу от монолитных решений, если они использовались,

на микросервисные решения с использованием языка Go.

Целью данной работы является изучение особенностей монолитной и микросервисной архитектур, а также реализация на языке Golang приложений, использующих эти архитектуры, для выявления преимуществ микросервисной архитектуры при использовании языка Go.

Для достижения данной цели были поставлены следующие задачи:

- Изучение синтаксиса, основных пакетов, а также модели конкурентности языка Go.
- Анализ монолитной архитектуры и изучение её преимуществ и недостатков.
- Анализ сервис-ориентированной и микросервисной архитектур. Изучение преимуществ и недостатков микросервисной архитектуры.
- Реализация на языке Golang приложений с монолитной и микросервисной архитектурами.
- Тестирование приложений на различных сценариях масштабирования.
- Сравнение результатов тестирования производительности приложений.

Научная новизна работы состоит в самой постановке задачи и заключается в том, что в отечественной литературе слабо отражена проблема сравнения производительности монолитной и микросервисной архитектур ввиду относительной новизны последней, в особенности реализованных с использованием языка Golang. Результаты работы могут использоваться для дальнейшего обоснования выбора той или иной архитектуры при проектировании новых веб-приложений.

Магистерская работа состоит из введения, пяти разделов, заключения, списка использованных источников. Общий объём работы – 63 страницы, из них 52 страницы – основное содержание, включая 13 рисунков и 4 таблицы, список использованных источников информации – 37 наименований.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Первый раздел «Язык Golang» посвящён обзору особенностям языка Golang, а именно синтаксиса языка, системы пакетов и модулей, а также конкурентной модели.

Язык Go, или Golang, является синтаксически схожим с C языком программирования, ввиду того, что разработчики по большей части основывались на язык C. Go является регистрозависимым языком с полной поддержкой символов Unicode. Идентификаторы могут представлять собой любую непустую последовательность символов, начинающуюся с буквы и не совпадающая ни с одним из ключевых слов языка.

В Go используется система пакетов. Каждая программа включает в себя один или несколько пакетов. Пакет, к которому относится данный файл исходного кода указывается в начале файла.

По умолчанию программа содержит один пакет main(), являющийся основным. Точкой входа в программу является функция main. Для предварительной инициализации определённых параметров программы могут быть использованы функции init(), выполняющиеся перед запуском main().

Стандартный пакет Golang содержит набор встроенных типов данных: целые числа, числа с плавающей запятой, символы, строки, логические значения, а также несколько специальных типов.

Целые числа могут быть представлены с помощью 11 целочисленных типов: целые числа со знаком и без знака фиксированного значения, знаковые и беззнаковые числа с размерностью, зависящей от архитектуры, специальные типы для работы с бинарными и символьными данными, и специальный беззнаковый тип, в котором битовая ширина совпадает с указателем.

Числа с плавающей точкой представлены типами с одинарной и двойной точности в соответствии со стандартом IEEE 754.

Для хранения коллекций данных используются массивы и срезы. Массивы представляют собой набор последовательно размещённых в памяти элементов одного типа с фиксированным размером. Срез является расширением над массивами, которое позволяет создавать коллекции данных динамической размерности. Основным отличием этих конструкция является то, что при работе с массивами данные передаются по значению, т.е. создаётся полная копия, а при работе со срезами – по ссылке.

Строки представляют собой байтовые срезы – срезы с целочисленными представлениями символов в кодировке Unicode.

На основе базовых типов можно реализовывать собственные с помощью структур и интерфейсов. Структуры содержат ссылки на структурированные коллекции данных различных типов. Интерфейсы же представляют собой абстрактные типы, определяющие множество функций, необходимых к реализации.

Конкурентность в Golang обеспечивается с помощью специальных сущностей – горутин и каналов. Горутины представляют собой легковесные сущности, конкурентно выполняемые в потоке приложения. Для обеспечения неблокирующей коммуникации между горутинами используется механизм каналов. Для обеспечения высокой эффективности вычислений, можно использовать конвейеризацию горутин, с помощью которой можно выполнять независящие друг от друга операции конкуррентно, и затем при необходимости передавать с помощью каналов в последующие горутины, выполняющую следующий этап обработки.

Во втором разделе «Архитектуры построения приложений» рассматриваются особенности монолитной, сервис-ориентированной и микросервисной архитектур и рассматриваются теоретические преимущества и недостатки архитектур.

Монолитная архитектура представляет собой подход к построению приложения, в котором весь код приложения находится в одном монолитном блоке и функции приложения работают в контексте этого блока. При построении монолитных приложений на языке Go, весь функционал приложения выполняется в едином процессе. Монолитная архитектура позволяет быстро создавать прототипы приложений и достаточно просто реализовывать функциональные изменения, так как всё приложение находится в одном месте и отслеживается единообразно. Однако, с увеличением размеров приложения, сложность его поддержки и масштабирования также увеличивается.

Преимуществами архитектуры являются простота понимания и реализации; при работе приложения между функциональными блоками будут минимальные задержки; приложение легко развернуть и масштабировать. Однако недостатками являются высокая связность при расширении функционала приложения; ввиду того, что приложение представляет собой один исполняемый

артефакт, масштабирование со временем будет мало эффективным; с ростом программы падает надёжность ввиду того, что при возникновении ошибки в одном блоке, станет недоступным всё приложение.

Сервис-ориентированная архитектура представляет собой подход, при котором отдельные функциональные блоки приложения разбиваются на отдельные сервисы. В СОА процессы, выполняющие схожие бизнес-задачи, объединяются в один крупный сервис, сервисы предприятия, а процессы, выполняющие общие для всех компонентов задачи, объединяются в более мелкие прикладные или инфраструктурные сервисы. Ввиду этого появляются уровни сервисов. Взаимодействие между сервисами происходим с помощью общей шины а также передачей контекста между сервисами.

В отличие от сервис-ориентированной архитектуры, микросервисная архитектура предполагает, что разные компоненты приложения скорее дублируют некоторый общий функционал, но не зависят от других компонентов. Достигается это путём разбиения системы на микросервисы по ограниченным областям бизнес-процессов.

Микросервис можно определить как маленький сервис, выполняющий только одну задачу. Они представляют собой небольшие, атомарные, выполняющие одну или реже несколько функций сервисы. Микросервисная архитектура разработки предлагает подход, в котором система разрабатывается как единое приложение, состоящее из нескольких небольших сервисов, каждый из которых выполняется в отдельном “процессе”. Коммуникация между сервисами осуществляется с помощью какого-либо механизма передачи данных. Микросервисы при этом развёртываются независимо друг от друга.

По сравнению с сервис-ориентированной архитектурой, у микросервисной отсутствуют уровни сервисов, т.е. все микросервисы равноправны; каждый микросервис выполняет строго одну определённую бизнес-задачу; для коммуникации между микросервисами редко используется передача контекста, но используется механизм подписки на обновления.

Преимуществами микросервисной архитектуры являются высокая организованность и слабая связноть микросервисов; легкость масштабирования; изоляция сбоев в одном микросервисе. К недостаткам же можно отнести сложность разработки, заключающуюся в организации способа коммуникации между сервисами и тестирования функционала конкретных микросервисов.

В третьем разделе «Разработка монолитного приложения» описывается процесс разработки на языке Golang монолитного приложения, а также кратко приводится информация об используемых при разработке технологий.

Для корректности дальнейшего проведения сравнения двух приложений, была спроектирована общая структура приложений. Для реализации было выбрано приложение кинотеатра, предоставляющего краткую информацию о киносеансах. В приложениях должны быть схожим образом реализован функционал предоставления данных пользователю с помощью вебсайта, а также функциональные блоки, предоставляющие данные о проводимых в сеансах: данные о пользователях, фильмах, сеансах и заказах.

При разработке монолитного приложения все запросы к приложению выполняются в рамках одного программного потока. Ввиду этого необходимо было выбрать подходящее средство для маршрутизации запросов. Ввиду предоставляемых возможностей по маршрутизации, был выбран фреймворк Gin. Основными преимуществами Gin являются высокая эффективность обработки запросов, поддержка связующих функциональных средств – middleware, а также возможность создания групповых маршрутов.

Для коммуникации с базой данных был выбран подход с использованием технологии объектно-реляционного отображения, или ORM. В качестве реализации данной технологии была выбрана библиотека GORM.

Для аутентификации пользователей используется технология JWT, позволяющая обмениваться данными между клиентом и сервером с помощью JSON-объектов, содержащих полезную нагрузку в закодированном виде и подпись.

Для контейнеризации и оркестрации приложения используются технологии Docker и Docker Compose соответственно, также реализованные на Golang.

Реализация приложения включала в себя разработку моделей сущностей базы данных и способа взаимодействия с ней, контроллеров запросов для обработки поступающих запросов и middleware для проведения промежуточных действий.

Модель сущности представляет собой структуру, хранящую в себе информацию об ID пользователя, имени, email, пароль и другие данные. При разработке модели был учтен тот факт, что при работе с базой данных будет использоваться технология ORM, и данные в структуре также имеют необходимое представление. Модели остальных сущностей были реализованы схожим образом.

Способ коммуникации с базой данных представляет собой установление соединения с существующей базой данных и последующее выполнение необходимых ORM-функций. При вызове данных функций они преобразуются в валидный SQL-запрос к базе данных.

Для авторизации пользователей была реализована промежуточная функция, принимающая на вход от информацию от пользователя, включающую в себя JWT-токен, и производящую проверку полученного токена на валидность. В случае, если токен не проходит проверку, то запрос отклоняется, иначе передаётся к следующему обработчику.

Запросы к конечным точкам приложения обрабатываются с помощью соответствующих этим точкам обработчикам. Маршрутизация запросов производится с помощью Gin и позволяет выполнять действия с данными в соответствии с типом запроса и полученной информации.

В четвёртом разделе «Разработка микросервисного приложения» описывается процесс разработки микросервисного приложения, включающий в себя также выбор способа коммуникации между микросервисами и распределения нагрузки на сервисы.

Для связи между сервисами был выбран протокол REST API. Выбор был основан на основании результатов исследований, показывающих, что при частой передаче сравнительно небольших объёмах данных, REST имеет преимущество над gRPC, другим часто используемым протоколом.

Для распределения нагрузки между сервисами была выбрана технология Traefic, также реализованная на Golang. Эта технология представляет собой обратный прокси-сервер с возможностью балансировки нагрузки как по четвёртому, так и по седьмому уровню OSI. Также Traefic позволяет сохранять установленную сессию между клиентом и конкретным микросервисом при кратковременном разрыве соединения.

В микросервисном приложении все функциональные блоки, реализованные в монолитном приложении, были подвержены процессу рефакторизации для преобразования их в отдельные несвязные микросервисы.

Данные о микросервисах, такие как их расположение, необходимое количество реплик, указываются в Docker compose файле, с помощью которого производится оркестрация микросервисов.

Реализация микросервиса вебсайта включала в себя разработку способа

подключения к репликам всех используемых микросервисов, а также обращение к необходимым микросервисам при получении запросов.

Микросервисы пользователей, фильмов, сеансов и заказов были реализованы с использованием общей базовой структуры микросервисов. Базовыми элементами данной структуры являются: способ базовой конфигурации микросервиса и установка обработчиков запросов.

Базовая конфигурация микросервисов включает в себя установку адреса микросервиса и соответствующей базы данных, а также маршрутизатора запросов к микросервису.

Функционал микросервисов был реализован таким образом, чтобы он повторял функционал соответствующего функционального блока монолитного приложения.

В пятом разделе «Сравнение разработанных приложений» описывается используемый способ тестирования приложений, а также приводятся результаты проведённого тестирования приложений.

Перед проведением тестирования выдвинуто предположение, что монолитная архитектура является более подходящей либо для небольших проектов, в которых нет какой-либо сложной логики и которые будут представлять собой лишь концепт решения, либо для больших и сложных решений, в которые не планируется добавлять новый функционал, либо очень редко, и которым важна высокая производительность и для которых гарантируется высокая стабильность.

Для проведения нагрузочных тестов приложений был выбран инструмент Gatling, позволяющий гибко настраивать нагрузку, используя конфигурационные файлы. Ввиду этого были разработаны скрипты, позволяющие выполнять тестирование по нескольким сценариям: при выполнении 30 пользователями по 1000 запросов к приложению, и по 10000 запросов. Также были разработаны скрипты, выполняющие запросы к приложениям в течение полутора и пятнадцати минут.

Анализируя результаты проведённого тестирования приложений по различным сценариям, было замечено, что приложение, использующее микросервисную архитектуру, оказалось более производительным и стабильным при масштабировании критических сервисов, чем аналогичное приложение, использующее монолитную архитектуру. Но в то же время ввиду того, что для масшта-

бирования необходимо выделять больше системных ресурсов, микросервисное решение может быть неактуальным для небольших проектов, не рассчитывающих на большой поток запросов. Общей же тенденцией является то, что с течением времени производительность приложений уменьшается на несколько процентов.

ЗАКЛЮЧЕНИЕ

В настоящее время всё больше разработчиков при разработке программного обеспечения выбирают архитектуры, направленные на модульность и переиспользуемость компонентов, несмотря на то что монолитная архитектура всё также является одной из самых стабильных. С ростом популярности языка Golang всё больше компаний начали процесс по переходу от монолитных решений, если они использовались, на микросервисные решения с использованием языка Go.

В настоящей работе были изучены синтаксис и конкурентная система языка Golang, изучены и проанализированы монолитная и микросервисная архитектура приложений. В последствие были реализованы монолитное и микросервисное приложение на языке Go. Также разработанные приложения были протестированы и сравнены их показатели производительности.

Golang является перспективным языком, позволяющим разрабатывать как монолитные, так и микросервисные приложения, эффективно выполняющие поставленные бизнес-задачи. Стоит отметить, что ввиду того, что язык Golang разрабатывался с оглядкой на уже существующие языки, в частности на языки программирования C и C++, разработчики могут достаточно просто использовать в новых проектах данный язык. Также немаловажной особенностью является то, что в данном языке реализована технология, позволяющая минимизировать такие ошибки приложений, как утечка памяти, состояние гонки и другие проблемы, возникающие в классических языках C и C++.

Приложение, использующее микросервисную архитектуру, в результате тестирования показалось более производительным и стабильным при масштабировании критических сервисов, чем аналогичное приложение, использующее монолитную архитектуру. Но в то же время ввиду того, что для масштабирования необходимо выделять больше системных ресурсов, микросервисное решение может быть неактуальным для небольших проектов, не рассчитывающих на большой поток запросов.

Основные источники информации:

1 2021 Developer Survey [Электронный ресурс]. — URL: `https://insights.stackoverflow.com/survey/2021` (Дата обращения 30.10.21). Загл. с экрана Яз. Англ.

2 Go by Example [Электронный ресурс]. — URL: https://gobyexample.com/ (Дата обращения 10.11.21). Загл. с экрана Яз. Англ.

3 *Anagnostopoulos, A.* Hands-On Software Engineering with Golang: Move beyond basic programming to design and build reliable software with clean code / A. Anagnostopoulos. — 1 edition. — Birmingham: Packt Publishing Limited, 2020. — Anagnostopoulos, Achilleas (VerfasserIn).

4 *Bob Strecansky,*. Hands-On High Performance with Go: Boost and optimize the performance of your Golang applications at Scale with Resilience / Bob Strecansky. — 1 edition. — Packt Publishing, 2020.

5 *Czarnul, P.* Parallel programming for modern high performance computing systems / P. Czarnul. A Chapman & Hall book. — Boca Raton and London and New York: CRC Press Taylor & Francis Group, 2018. — Czarnul, Paweł (VerfasserIn).

6 *Katherine Cox-Buday,*. Concurrency in Go: Tools and Techniques for Developers / Katherine Cox-Buday. — 1 edition. — O'Reilly Media, 2017.

7 *Mario Castro Contreras,*. Go Design Patterns / Mario Castro Contreras. — Birmingham: Packt Publishing, 2017.

8 *Oliinik, Olena and Avdieiev, Oleksii and Freher, Oleh,*. Golang pros and cons: Why use golang for your project / Oliinik, Olena and Avdieiev, Oleksii and Freher, Oleh // *InterConf*. — 2020.

9 *Tsoukalos, M.* Mastering Go: Create Golang production applications using network libraries, concurrency, and advanced Go data structures / Mihalis Tsoukalos / M. Tsoukalos. — Second edition edition. — Birmingham: Packt Publishing Ltd, 2019.

10 *Whitney, J.* Distributed execution of communicating sequential process-style concurrency: Golang case study / J. Whitney, C. Gifford, M. Pantoja // *The Journal of Supercomputing*. — 2019. — Vol. 75, no. 3. — Pp. 1396–1409.

11 *De Lauretis, L.* From monolithic architecture to microservices architecture // 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) / IEEE. — 2019.

12 *Ponce, F.* Migrating from monolithic architecture to microservices: A rapid review // 2019 38th International Conference of the Chilean Computer Science Society (SCCC) / IEEE. — 2019.

13 *Villamizar, M.* Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud // 2015 10th Computing Colombian Conference (10CCC) / IEEE. — 2015.

14 *Mendonca, N. C.* The monolith strikes back: Why istio migrated from microservices to a monolithic architecture / N. C. Mendonca, C. Box, C. Manolache, L. Ryan // *IEEE Software*. — 2021. — Vol. 38, no. 05.

15 *Yellavula, N.* Building RESTful Web services with Go: Learn how to build powerful RESTful APIs with Golang that scale gracefully / N. Yellavula. — Packt Publishing Ltd, 2017.

16 *Jones, M.* Json web token (jwt) profile for oauth 2.0 client authentication and authorization grants / M. Jones, B. Campbell, C. Mortimore // *May-2015.{Online}. Available: https://tools. ietf. org/html/rfc7523*. — 2015.

17 *Rad, B. B.* An introduction to docker and analysis of its performance / B. B. Rad, H. J. Bhatti, M. Ahmadi // *International Journal of Computer Science and Network Security (IJCSNS)*. — 2017. — Vol. 17, no. 3.

18 *Jangla, K.* Docker compose / K. Jangla // Accelerating Development Velocity Using Docker. — Springer, 2018. — Pp. 77–98.

19 *Homay, A.* A survey: Microservices architecture in advanced manufacturing systems // 2019 IEEE 17th International Conference on Industrial Informatics (INDIN) / IEEE. — Vol. 1. — 2019. — Pp. 1165–1168.

20 *Cerquitelli, T.* Manufacturing as a data-driven practice: methodologies, technologies, and tools / T. Cerquitelli, D. J. Pagliari, A. Calimera, L. Bottaccioli, E. Patti, A. Acquaviva, M. Poncino // *Proceedings of the IEEE*. — 2021. — Vol. 109, no. 4. — Pp. 399–422.

21 *Yang, H.* Microservices-based cloud-edge collaborative condition monitoring platform for smart manufacturing systems / H. Yang, S. Ong, A. Nee, G. Jiang, X. Mei // *International Journal of Production Research*. — 2022. — Pp. 1–10.

22 *Al-Debagy, O.* A comparative review of microservices and monolithic architectures // 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI) / IEEE. — 2018. — Pp. 000149–000154.

23 *Pachghare, V. K.* Microservices Architecture for Cloud Computing / V. K. Pachghare // *architecture*. — 2016. — Vol. 3. — P. 4.

24 *Baboi, M.* Dynamic microservices to create scalable and fault tolerance architecture / M. Baboi, A. Iftene, D. Gîfu // *Procedia Computer Science*. — 2019. — Vol. 159. — Pp. 1035–1044.

25 *Liu, G.* Microservices: architecture, container, and challenges // 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C) / IEEE. — 2020. — Pp. 629–635.

26 Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems / M. Bolanowski, K. Żak, A. Paszkiewicz, M. Ganzha, M. Paprzycki, P. Sowiński, I. Lacalle, C. E. Palau // *arXiv preprint arXiv:2208.00682*. — 2022.

27 *Indrasiri, K.* Microservices for the Enterprise: Designing, Developing, and Deploying / K. Indrasiri, P. Siriwardena // *Apress, Berkeley*. — 2022.

28 *Chamas, C. L.* Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis // 2017 IEEE 9th Latin-American Conference on Communications (LATINCOM) / IEEE. — 2017. — Pp. 1–6.

29 *Valkov, I.* Comparing languages for engineering server software: Erlang, Go, and Scala with Akka // Proceedings of the 33rd Annual ACM Symposium on Applied Computing. — 2018. — Pp. 218–225.

30 *Irro, M.* Concurrent programming with actors and microservices / M. Irro // *Evaluation*. — 2018. — Vol. 5. — P. 4.

31 *Johansson, A.* HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm. — 2022.

32 *Sharma, R.* Traefik API Gateway for Microservices / R. Sharma, A. Mathur.

33 Traefik Proxy. — URL: `https://doc.traefik.io/traefik/` (Дата обращения 08.12.22). Загл с экрана Яз. Англ.

34 *Kalske, M.* Challenges when moving from monolith to microservice architecture // Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17 / Springer. — 2018. — Pp. 32–47.

35 *Stubbs, J.* Distributed systems of microservices using docker and serfnode // 2015 7th International Workshop on Science Gateways / IEEE. — 2015. — Pp. 34–39.

36 *Saransig, A.* Performance analysis of monolithic and micro service architectures–containers technology // Trends and Applications in Software Engineering: Proceedings of the 7th International Conference on Software Process Improvement (CIMPS 2018) 7 / Springer. — 2019. — Pp. 270–279.

37 *Freire, A. F. A.* Migrating production monolithic systems to microservices using aspect oriented programming / A. F. A. Freire, A. F. Sampaio, L. H. L. Carvalho, O. Medeiros, N. C. Mendonça // *Software: Practice and Experience*. — 2021. — Vol. 51, no. 6. — Pp. 1280–1307.