МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ

ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»



Направление подготовки   
09.03.01 Информатика и вычислительная техника

Направленность (профиль)

«Технологии разработки программного обеспечения»

**Выпускная квалификационная работа**

Оптимизация выполнения serverless-функций на облачных инфраструктурах на основе импортозамещения

|  |
| --- |
| Обучающегося 4 курса  очной формы обучения  Величко Арсения Александровича |
|  |
| Руководитель выпускной квалификационной работы:  кандидат педагогических наук, доцент, Илья Борисович Государев |
|  |

Санкт-Петербург

2025

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc200062607)

[ВВЕДЕНИЕ 3](#_Toc200062608)

[ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 6](#_Toc200062609)

[Понятие serverless вычислений 6](#_Toc200062610)

[Serverless функции 8](#_Toc200062611)

[Оптимизация serverless функций 11](#_Toc200062612)

[Импортозамещение в сфере облачной инфраструктуры 14](#_Toc200062613)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 19](#_Toc200062614)

[Программный продукт 19](#_Toc200062615)

[Выбор инструментов и технологий для разработки 24](#_Toc200062616)

[Реализация фронтенда веб-приложения 28](#_Toc200062617)

[Реализация бекенда веб-приложения 33](#_Toc200062618)

[ЗАКЛЮЧЕНИЕ 41](#_Toc200062619)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 43](#_Toc200062620)

[ПРИЛОЖЕНИЯ 46](#_Toc200062621)

# ВВЕДЕНИЕ

**Актуальность работы** обусловлена существующим противоречием между постоянно возрастающей потребностью отечественных организаций в оптимизации производительности и стоимости эксплуатации serverless-функций на национальной облачной инфраструктуре и отсутствием специализированных инструментов и приложений, способных эффективно решать данные задачи. В настоящее время активно развивается процесс импортозамещения, при котором компании и государственные учреждения вынуждены переходить с зарубежных облачных решений на отечественные платформы, включая Yandex Cloud Functions [1]. В связи с этим возникает необходимость в способах оперативного переноса и оптимизации ранее разработанных функций, обеспечивающих высокий уровень производительности и приемлемый уровень затрат на облачные вычислительные ресурсы. При этом стандартные подходы к оптимизации serverless-функций, основанные на ручном анализе и модификации исходного кода, не позволяют быстро и качественно решать задачи, возникающие при миграции большого количества приложений и сервисов.

Вместе с тем на сегодняшний день отсутствуют отечественные программные продукты, предоставляющие комплексные и автоматизированные решения для анализа, оптимизации и замера производительности serverless-функций, которые учитывают специфику национальных облачных платформ и обеспечивают удобные механизмы взаимодействия разработчиков с процессом улучшения производительности. Отсутствие таких инструментов негативно отражается на скорости и эффективности процесса переноса и запуска функций, увеличивая временные и финансовые затраты компаний, вынужденных вручную проводить работы по адаптации и улучшению своего кода. Кроме того, ручная оптимизация кода сопряжена с высоким риском ошибок и недостаточной точностью оценки эффективности произведенных изменений, что снижает надежность и устойчивость систем.

**Цель** данной работы заключается в разработке программного комплекса для автоматизированного запуска, анализа и оптимизации производительности serverless-функций, размещенных на отечественной облачной инфраструктуре. Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Провести анализ существующих подходов к реализации и оптимизации serverless-функций, изучить ключевые характеристики и метрики производительности, актуальные для данных технологий.
2. Изучить специфику национальной облачной платформы Yandex Cloud Functions, выявить отличия и особенности работы с ней по сравнению с зарубежными аналогами, в том числе AWS Lambda.
3. Разработать архитектуру и выбрать подходящие инструменты и технологии для реализации серверной и клиентской частей программного продукта, позволяющего автоматизировать процессы оптимизации и мониторинга serverless-функций.
4. Реализовать модули автоматизированной оптимизации кода с помощью больших языковых моделей, а также инструментов анализа и визуализации метрик производительности serverless-функций.
5. Разработать удобный пользовательский интерфейс, обеспечивающий возможность управления, тестирования и анализа функций в режиме реального времени с представлением визуализированных результатов и рекомендаций.
6. Провести тестирование и проверку работоспособности разработанного программного продукта на примере реальных serverless-функций, размещенных на платформе Yandex Cloud Functions, и подтвердить эффективность предложенного решения.

Предполагаемый **результат работы** — программный комплекс, состоящий из серверной части и веб-интерфейса, предназначенный для автоматизированного запуска и анализа производительности serverless-функций на облачной инфраструктуре, реализованной в рамках политики импортозамещения. Данный комплекс включает встроенный редактор кода, модуль автоматического сбора и обработки метрик производительности, а также инструмент для сравнения показателей до и после проведения оптимизации кода. Код функций будет передаваться на обработку в большие языковые модели, которые предложат оптимизированный вариант программного кода и сформулируют конкретные рекомендации по улучшению производительности. Полученные результаты и рекомендации будут визуализироваться через веб-портал, предоставляющий пользователям возможность детального анализа метрик производительности и принятия взвешенных решений по конфигурации и запуску оптимизированных функций.

Разработанный программный продукт обеспечит решение проблемы, связанной с недостаточной эффективностью и трудоемкостью ручной оптимизации serverless-функций, ускорит процессы их миграции и эксплуатации на национальной облачной платформе, снизит затраты на облачные вычисления и повысит общую производительность информационных систем организаций. Применение данного решения также позволит компаниям и государственным учреждениям успешно реализовать политику импортозамещения в сфере информационных технологий, сократить зависимость от зарубежных решений и обеспечить высокий уровень надежности и безопасности собственных сервисов.

# ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## Понятие serverless вычислений

Понятие «serverless-вычисления» означает облачную модель, при которой вся инфраструктура управляется провайдером, а пользователю не нужно заниматься настройкой и обслуживанием серверов или операционных систем. Несмотря на терминологию, подразумевающую отсутствие серверов, физические или виртуальные серверы продолжают существовать, однако их наличие и состояние абстрагируются от конечного пользователя, становясь полностью прозрачными с точки зрения выполнения программного обеспечения. Это означает, что пользователи взаимодействуют исключительно с высокоуровневыми механизмами предоставления ресурсов, которые активируются по запросу и оплачиваются пропорционально фактическому времени их использования [2].

Основополагающей характеристикой serverless-вычислений является событийно-ориентированная модель исполнения кода, согласно которой вычислительные ресурсы выделяются облачным провайдером динамически, в зависимости от текущей нагрузки и поступающих запросов. В этом заключается принципиальное отличие подхода serverless от традиционных подходов к организации вычислений, таких как виртуализация и контейнеризация, где ресурсы обычно резервируются заранее, исходя из прогнозируемой нагрузки и периодически корректируемых сценариев эксплуатации. В serverless-подходе инфраструктура автоматически адаптируется к текущим требованиям, исключая необходимость ручного управления мощностями и обеспечивая оптимальное соответствие выделяемых ресурсов запросам пользователей в реальном времени.

Облачная среда, являющаяся основой для serverless-вычислений, представляет собой совокупность распределенных вычислительных мощностей, управляемых специализированным программным обеспечением. Это программное обеспечение отвечает за создание и уничтожение контейнеров или других форм легковесных виртуальных сред, в которых осуществляется непосредственное выполнение пользовательских задач. Каждый запуск такого контейнера инициируется возникновением определенного события — например, поступлением сетевого запроса, изменения в хранилище данных или прошествие заданного временного интервала. Облачный провайдер, контролирующий инфраструктуру, берет на себя задачу мониторинга, балансировки нагрузки, распределения ресурсов и изоляции выполняемых приложений друг от друга.

Оплата использования ресурсов в рамках serverless-модели осуществляется по принципу «плати за потребленное», то есть пользователь оплачивает только фактически использованное процессорное время и оперативную память в момент выполнения конкретных задач. В отличие от традиционной инфраструктуры или инфраструктуры на основе виртуальных машин и контейнеров, при использовании которых стоимость аренды вычислительных мощностей рассчитывается исходя из фиксированных тарифов за период времени, serverless-подход предполагает посекундный или даже миллисекундный расчет стоимости. Такой способ тарификации позволяет минимизировать затраты при неравномерной или непредсказуемой нагрузке, что является значительным преимуществом при проектировании систем с переменными или непостоянными вычислительными потребностями.

При этом следует учитывать, что модель serverless-вычислений имеет определенные особенности, накладывающие ограничения на спектр ее применения. В частности, serverless-вычисления плохо подходят для длительных или постоянных нагрузок, требующих непрерывного выполнения кода и поддержания долгосрочного состояния внутри среды исполнения. Кроме того, возможность точной настройки инфраструктуры и управления низкоуровневыми параметрами вычислительной среды сильно ограничена, поскольку облачный провайдер полностью контролирует распределение и конфигурацию ресурсов. Важным аспектом является и ограничение времени и ресурсов, предоставляемых на каждую отдельную задачу, что также может существенно ограничивать область применения serverless-технологий. Верхнеуровневое сравнение традиционной модели облачных вычислений и serverless представлено в таблице 1.

Таблица 1. Сравнение традиционных и serverless вычислений

| **Критерий** | **Serverless-вычисления** | **Традиционная модель** |
| --- | --- | --- |
| Управление инфраструктурой | Полностью перенесено к облачному провайдеру | Ответственность пользователя |
| Масштабирование | Автоматическое реактивное под нагрузкой | Ручная или полуавтоматическая настройка |
| Модель оплаты | За фактическое время исполнения кода | За выделенные вычислительные мощности |
| Время развертывания | Секунды до минут | Минуты до часов |
| Контроль над окружением | Ограничен, доступ лишь к высокоуровневым параметрам | Полный, включая операционную систему |
| Поддержка долгоживущих процессов | Ограничена заданными тайм-аутами | Не ограничена, возможна постоянная работа |
| Чувствительность к «холодному старту» | Высокая, влияет на задержку ответа | Отсутствует |
| Типовые сценарии | Событийная обработка, нерегулярные нагрузки | Длительные задачи, предсказуемая нагрузка |
| Экономическая эффективность при пиковой нагрузке | Выше за счёт эластичности | Зависит от резервирования ресурсов |

## Serverless функции

Serverless-технологии возникли в результате стремления разработчиков и организаций к минимизации расходов на инфраструктуру и сокращению временных затрат на управление вычислительными ресурсами. Первым этапом в истории развития данного подхода стало появление облачных вычислений, которое обеспечило возможность переноса серверной инфраструктуры в удаленные центры обработки данных. Следующим шагом стал переход от виртуальных машин и контейнеров к еще более абстрактной модели, где исполнение кода осуществляется в среде, полностью управляемой поставщиком облачных услуг. Первым известным коммерческим решением данного типа стала платформа AWS Lambda от компании Amazon [3], представленная в 2014 году. Вскоре аналогичные решения появились и у других крупных компаний, таких как Microsoft (Azure Functions), Google (Google Cloud Functions), IBM (IBM Cloud Functions), а также на российском рынке, например, Yandex Cloud Functions.

Serverless функции являются автономными единицами кода, исполняемыми в среде поставщика облачных вычислений. Они состоят из нескольких ключевых элементов: непосредственно программного кода, определённой точки входа в приложение и конфигурации, задающей параметры выполнения функции. Технически serverless-функции могут быть написаны на широком спектре языков программирования, среди которых чаще всего встречаются Python, JavaScript, Java, C#, Ruby и Go. Поддержка языков определяется конкретным облачным провайдером, однако практически каждый из них предоставляет разработчикам возможность использования наиболее популярных и востребованных языков программирования.

Архитектурно serverless-функции встраиваются в существующие информационные системы посредством механизмов событийного взаимодействия и использования API. Это означает, что запуск функций происходит не постоянно, а только в ответ на определенные события, такие как HTTP-запросы, загрузка файлов, обновление баз данных, поступление сообщений в очередь или наступление таймерных событий. Функции выполняются в краткосрочном окружении, автоматически запускаемом и завершаемом облачным провайдером, который берет на себя вопросы масштабирования и распределения вычислительной нагрузки.

Наиболее распространённые способы применения serverless-функций включают автоматизацию обработки данных, генерацию ответов в веб- и мобильных приложениях, интеграцию различных сервисов, работу с мультимедиа и построение микросервисной архитектуры. Благодаря событийной активации функций, они особенно эффективны в задачах, которые возникают нерегулярно и не требуют постоянной нагрузки на вычислительные ресурсы.

Применение serverless-архитектуры, в отличие от традиционных подходов к размещению и запуску приложений, даёт преимущества в виде оптимизации затрат и упрощённого администрирования. Основным преимуществом является модель оплаты «за фактическое использование», при которой пользователи платят только за вычислительное время выполнения функций, не оплачивая простои или резервирование ресурсов. Кроме того, автоматическое масштабирование, предоставляемое облачными платформами, исключает необходимость предварительного планирования вычислительных мощностей и позволяет мгновенно адаптироваться к изменениям нагрузки.

Несмотря на указанные преимущества, serverless-функции имеют также ряд ограничений и недостатков, которые необходимо учитывать при проектировании и разработке приложений. Одним из ключевых ограничений является время «холодного старта», обусловленное запуском функции после периода бездействия, которое приводит к увеличению задержек при обработке запросов. Кроме того, serverless функции обычно имеют жесткие ограничения на время выполнения, объем выделяемой оперативной памяти и размер загружаемого кода, что накладывает серьезные ограничения на тип задач, для которых возможно их эффективное применение. С точки зрения архитектуры, serverless-технологии также могут приводить к проблемам сложности управления и мониторинга распределенных систем, состоящих из большого числа автономных компонентов.

Прием и передача данных в рамках исполнения serverless-функций осуществляется через стандартные механизмы и форматы, наиболее распространенными из которых являются JSON и XML. Функции получают данные через параметры вызова, передаваемые из источника события, например, от HTTP-запроса или сообщения из очереди. Результатом выполнения функции обычно также является структурированный ответ, возвращаемый в виде JSON-объекта или аналогичного формата, что позволяет эффективно интегрировать функции с внешними системами и сервисами.

Особенности разработки приложений с использованием serverless-функций включают необходимость учитывать кратковременный, лишенный состояния характер исполнения кода. Для работы с постоянными данными в serverless-архитектуре принято использовать внешние хранилища данных, такие как базы данных, очереди сообщений и облачные файловые системы. Разработка функции serverless также требует применения специализированных инструментов для локального тестирования и развертывания, таких как Serverless Framework, AWS SAM или Azure Functions Core Tools, что позволяет ускорить цикл разработки и снизить вероятность возникновения ошибок в продакшен-среде. Еще одним важным аспектом является мониторинг производительности функций и обработка возникающих исключений, для чего обычно используются специализированные средства мониторинга, предлагаемые облачными платформами или сторонними производителями.

## Оптимизация serverless функций

Оптимизация serverless-функций представляет собой комплекс мероприятий, направленных на повышение эффективности их исполнения посредством управления ключевыми метриками производительности. Наиболее значимыми показателями, подлежащими оптимизации, выступают потребление процессорных ресурсов, используемый объем оперативной памяти, общее время выполнения кода, время запуска функции и объем сетевого трафика. В каждом случае оптимизация преследует достижение баланса между минимальными затратами ресурсов и максимальной производительностью, что особенно актуально в условиях поминутной или посекундной оплаты облачных вычислений.

Оптимизация потребления процессорных ресурсов serverless-функций достигается за счет эффективного алгоритмического решения задач и применения более производительных языков и технологий разработки. Например, переписывание ресурсоемких участков кода на компилируемые языки или языки с низким уровнем абстракции может существенно сократить количество процессорных циклов, затрачиваемых на исполнение кода. Кроме того, уменьшение операций ввода-вывода и обращений к внешним сервисам способствует снижению общей нагрузки на процессор, что улучшает экономическую эффективность serverless-решений.

Уменьшение потребляемой оперативной памяти является еще одной важной задачей оптимизации функций. Данный процесс требует контроля за утечками памяти, эффективного управления жизненным циклом объектов и строгой оптимизации используемых структур данных. Также положительное влияние оказывает выбор подходящих библиотек, минимизирующих внутренние расходы памяти. Снижение объемов памяти, запрашиваемой функцией, позволяет существенно сократить затраты на облачные вычисления и одновременно увеличить общую пропускную способность системы за счет возможности параллельного выполнения большего числа задач.

Оптимизация времени выполнения задач подразумевает сокращение общей продолжительности обработки конкретного запроса. Для достижения этого результата используются методы предварительного вычисления часто запрашиваемых данных, кеширование промежуточных результатов и асинхронная обработка задач, не требующих немедленного ответа. Кроме того, важным фактором является правильный выбор конфигурации запускаемых функций, включающий распределение задач на отдельные модули, которые могут быть исполнены параллельно, а также рациональное использование выделенных вычислительных ресурсов.

Сокращение времени запуска функции является одним из наиболее сложных направлений оптимизации serverless-технологий, поскольку оно напрямую связано с архитектурой инфраструктуры поставщика облачных услуг. Для минимизации задержек, связанных с холодным запуском функций, применяют такие техники, как поддержание функции в активном состоянии путем регулярного ее вызова или предварительная инициализация некоторых ресурсов во время первого запуска. Несмотря на дополнительные расходы ресурсов, подобные методы позволяют существенно сократить время отклика и улучшить пользовательский опыт при работе с системой.

Еще одним направлением является оптимизация сетевого трафика, которая достигается путем уменьшения объема данных, передаваемых между функцией и внешними сервисами. Для этого используются алгоритмы сжатия, минимизация форматов передачи данных и применение эффективных протоколов коммуникации, таких как HTTP/2 или gRPC. Оптимизация сетевого взаимодействия позволяет не только снизить прямые расходы на трафик, но и улучшить время отклика системы в целом.

В последние годы широкое распространение получили методы оптимизации serverless-функций с применением больших языковых моделей, также известных как LLM. Эти модели позволяют автоматизировать ряд задач по повышению эффективности функций, включая авторефакторинг программного кода и генерацию рекомендаций для разработчиков по улучшению производительности. [4]

Авторефакторинг кода с использованием LLM подразумевает автоматическое преобразование исходного кода с целью устранения неэффективных участков и внедрения оптимизированных конструкций. Большие языковые модели, обученные на обширных массивах программного кода, способны выявлять шаблоны неоптимального использования ресурсов и предлагать альтернативные решения, которые уменьшают время исполнения задач и снижают нагрузку на ресурсы вычислительной среды. Применение авторефакторинга позволяет не только повысить производительность serverless-функций, но и сократить затраты времени разработчиков на рутинные процессы оптимизации.

Другим направлением применения LLM является автоматическая генерация текстовых рекомендаций по улучшению производительности функций. На основе анализа кода и результатов его исполнения большие языковые модели способны формировать персонализированные и контекстно обусловленные советы, касающиеся более эффективного использования структур данных, сокращения числа запросов к внешним ресурсам и других аспектов работы приложений. В результате, применение LLM значительно упрощает процесс выявления узких мест в производительности и предоставляет разработчикам конкретные указания, позволяющие оперативно улучшить работу serverless-функций.

## Импортозамещение в сфере облачной инфраструктуры

Импортозамещение представляет собой комплекс мероприятий и стратегических решений, направленных на замещение продукции и услуг иностранного происхождения аналогичными товарами и услугами, производимыми на территории конкретного государства. Концепция импортозамещения исторически связана с необходимостью достижения национальной экономической безопасности и снижения зависимости от внешних рынков, особенно в условиях экономических кризисов, санкционных ограничений и геополитической нестабильности. Подобный подход позволяет снизить риски, возникающие при перебоях в международной торговле, и создать условия для устойчивого развития внутреннего производства. Практическое применение политики импортозамещения охватывает широкий спектр отраслей, включая промышленность, сельское хозяйство, энергетику, здравоохранение и информационные технологии.

Актуальность политики импортозамещения обусловлена необходимостью минимизации последствий экономических санкций и нестабильности на международных рынках. В условиях сокращения возможностей импорта ключевых продуктов и услуг, государство и предприятия вынуждены оперативно переориентироваться на внутренние ресурсы. Поддержка отечественного производителя способствует развитию собственной промышленной базы, стимулирует инновационную деятельность и создает дополнительные рабочие места. Следовательно, импортозамещение служит важным инструментом повышения национального экономического потенциала и обеспечения стабильности в условиях внешнеэкономической неопределенности.

В сфере информационных технологий и, в частности, облачной инфраструктуры, политика импортозамещения выражается в развитии национальных провайдеров облачных услуг. Российский рынок облачных вычислений начал активно формироваться в 2000-х годах, однако наиболее динамичное развитие национальных облачных провайдеров произошло в период после 2014 года, в ответ на санкционные ограничения, затронувшие сектор информационных технологий. В результате в России начался активный процесс перехода от зарубежных решений к собственным, разработанным и размещенным внутри страны.

Одним из крупнейших отечественных провайдеров облачной инфраструктуры является компания «Яндекс». Проект облачной платформы «Yandex Cloud» был запущен в 2018 году, став одной из наиболее масштабных отечественных инициатив в этой области [5]. Платформа предоставляет пользователям широкий спектр сервисов, включая виртуальные машины, хранилища данных, серверы баз данных, сервисы машинного обучения и искусственного интеллекта [6]. В последующие годы компания активно расширяла портфель облачных продуктов, что позволило ей занять значительную долю на рынке отечественных облачных услуг.

Также значительный вклад в развитие национальной облачной инфраструктуры вносят компании Selectel и SberCloud, предлагающие различные уровни облачных решений, от инфраструктуры как сервиса до специализированных облачных сервисов в области искусственного интеллекта и анализа данных. Каждый из указанных провайдеров активно участвует в реализации программ импортозамещения, способствуя снижению зависимости российских предприятий и государственных структур от зарубежных технологических решений.

На российском рынке облачных технологий поддержку serverless-функций предоставляют несколько ключевых провайдеров. Компания Yandex Cloud запустила сервис Yandex Cloud Functions в 2018 году, предложив пользователям возможность выполнять код в отказоустойчивой и масштабируемой среде без необходимости управления виртуальными машинами. Selectel представила собственный сервис «Serverless: облачные функции» в январе 2020 года [7], основанный на платформе Apache OpenWhisk, расширяя возможности своей облачной платформы. SberCloud, в партнерстве с Huawei, внедрила сервис FunctionGraph в рамках своей платформы SberCloud.Advanced, предоставляя пользователям инструменты для разработки и запуска бессерверных приложений [8].

Старейшей и наиболее популярной платформой serverless функций в мире является AWS Lambda. Наиболее успешным отечественным аналогом этой платформы по праву можно считать Yandex Cloud Functions. Благодаря этому, именно на него часто падает выбор компаний, желающих перенести свои продукты на отечественную инфраструктуру. Рассмотрим основные отличия между этими двумя платформами.

При сравнении платформ Yandex Cloud Functions и AWS Lambda необходимо учитывать несколько ключевых аспектов, отражающих различия как на техническом, так и на организационном уровнях. Первым существенным моментом является набор поддерживаемых языков программирования и доступных сред исполнения. AWS Lambda предоставляет широкий спектр сред исполнения, включая Node.js, Python, Java, C#, Go, Ruby, а также пользовательские среды исполнения, позволяющие использовать практически любой язык программирования. В то же время Yandex Cloud Functions, хоть и поддерживают многие популярные языки, такие как Node.js, Python, Go, Java, C#, PHP, Bash и R, не предлагает встроенного механизма для создания полностью кастомных сред исполнения. Таким образом, при переносе функций с AWS Lambda на Yandex Cloud может потребоваться изменение используемого языка или дополнительная адаптация кода и зависимостей под специфику доступных сред.

Другой важный аспект связан с механизмами триггеров и интеграции с внешними сервисами. AWS Lambda глубоко интегрирована в экосистему AWS, включая сервисы Amazon S3, DynamoDB, Kinesis, API Gateway и CloudWatch Events. Это позволяет эффективно реализовывать сложные распределенные архитектуры, состоящие из множества взаимодействующих сервисов. Yandex Cloud Functions также обеспечивает интеграцию с другими сервисами своей облачной платформы, такими как Object Storage, Message Queue, Yandex API Gateway и Cloud Logs. Однако специфика работы и конфигурации триггеров может существенно отличаться. В связи с этим при переносе функций может потребоваться пересмотр архитектуры, включая переработку логики обработки событий и взаимодействия между компонентами системы.

Также необходимо учитывать разницу в подходах к мониторингу и логированию выполнения serverless-функций. AWS Lambda предоставляет детализированные средства мониторинга через Amazon CloudWatch, позволяющие отслеживать метрики выполнения функций, задержки, количество ошибок и другие ключевые параметры в реальном времени. При этом сбор и анализ логов в AWS осуществляется через интегрированный механизм, поддерживающий фильтрацию и поисковые запросы. Yandex Cloud Functions использует собственную систему мониторинга и журналирования событий, имеющую специфический интерфейс и набор возможностей [9]. Таким образом, перенос функций с AWS Lambda потребует адаптации существующих практик мониторинга и анализа журналов под новые инструменты и их возможности.

Наконец, важно обратить внимание на различия в подходах к управлению ресурсами и масштабированию. AWS Lambda обеспечивает достаточно гибкие механизмы автоматического масштабирования, позволяющие динамически выделять ресурсы в зависимости от нагрузки и управлять параллелизмом исполнения функций. Yandex Cloud Functions также реализует автоматическое масштабирование, однако алгоритмы распределения нагрузки и ограничения параллельных исполнений могут отличаться. В этом контексте перенос функций может потребовать дополнительного тестирования и настройки параметров выполнения, чтобы обеспечить ожидаемую производительность и эффективность использования ресурсов.

# ПРАКТИЧЕСКАЯ ЧАСТЬ

## Программный продукт

Разрабатываемый программный продукт представляет собой комплекс, предназначенный для автоматизированного запуска и анализа производительности serverless-функций, размещённых на отечественной облачной инфраструктуре, направленный на решение задачи повышения эффективности и оптимизации работы данных функций. Программный комплекс состоит из двух взаимосвязанных компонентов: серверной части и веб-интерфейса, предоставляющего пользователям инструменты для редактирования кода функций, управления их версиями, запуска тестирования и визуализации полученных результатов. В основе данного продукта лежит интеграция с большими языковыми моделями, осуществляющими автоматический анализ и улучшение исходного кода функций, что позволяет пользователям получать рекомендации по повышению производительности их приложений.

Функциональная часть продукта реализует несколько ключевых сценариев взаимодействия с системой. Одним из таких сценариев является создание и управление serverless-функциями, где разработчик получает возможность не только создавать новые функции, но и вести контроль за их версиями, обеспечивать систематизацию своего кода. Разработчик инициирует процесс создания новой функции посредством специального редактора кода, вводя необходимые атрибуты, такие как название функции, выбор языка программирования и платформы исполнения. В результате совершения этих действий происходит создание новой записи с сохранением введённых данных, и пользователь получает возможность последующего просмотра и редактирования созданного программного кода. Принцип работы данного сценария отражен на диаграмме последовательности (рисунок 1).

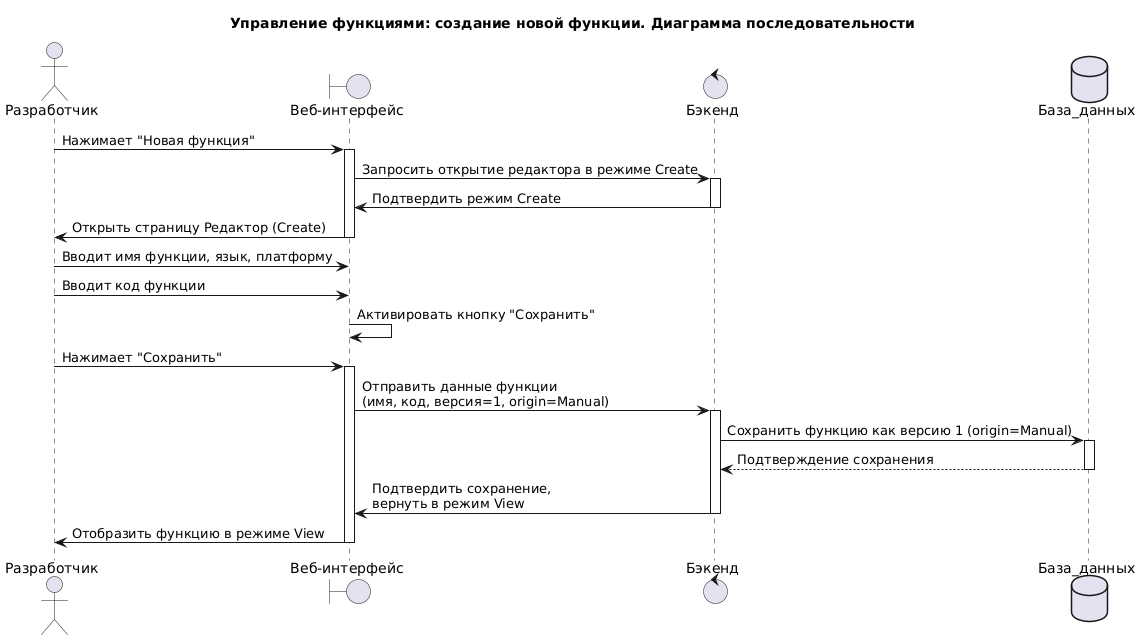


Рис. 1. Управление функциями: создание новой функции. Диаграмма последовательности

Следующим важным элементом, реализованным в продукте, является возможность автоматизированной оптимизации функций. Разработчик, используя специальный интерфейс, запускает процесс оптимизации исходного кода функции, который передаётся на обработку в большие языковые модели таких производителей как OpenAI и Anthropic. В ответ система получает улучшенную версию кода и подробное текстовое описание внесённых изменений, включающее информацию о целевых метриках оптимизации. Оптимизированный код сохраняется в системе в виде новой версии, доступной разработчику для последующего анализа, использования и редактирования. Помимо автоматизированной оптимизации предусмотрена возможность ручной корректировки кода, что позволяет разработчику вносить дополнительные улучшения на основе личного опыта и знаний специфики проекта. Для демонстрации данного сценария была подготовлена диаграмма последовательности (рисунок 2).

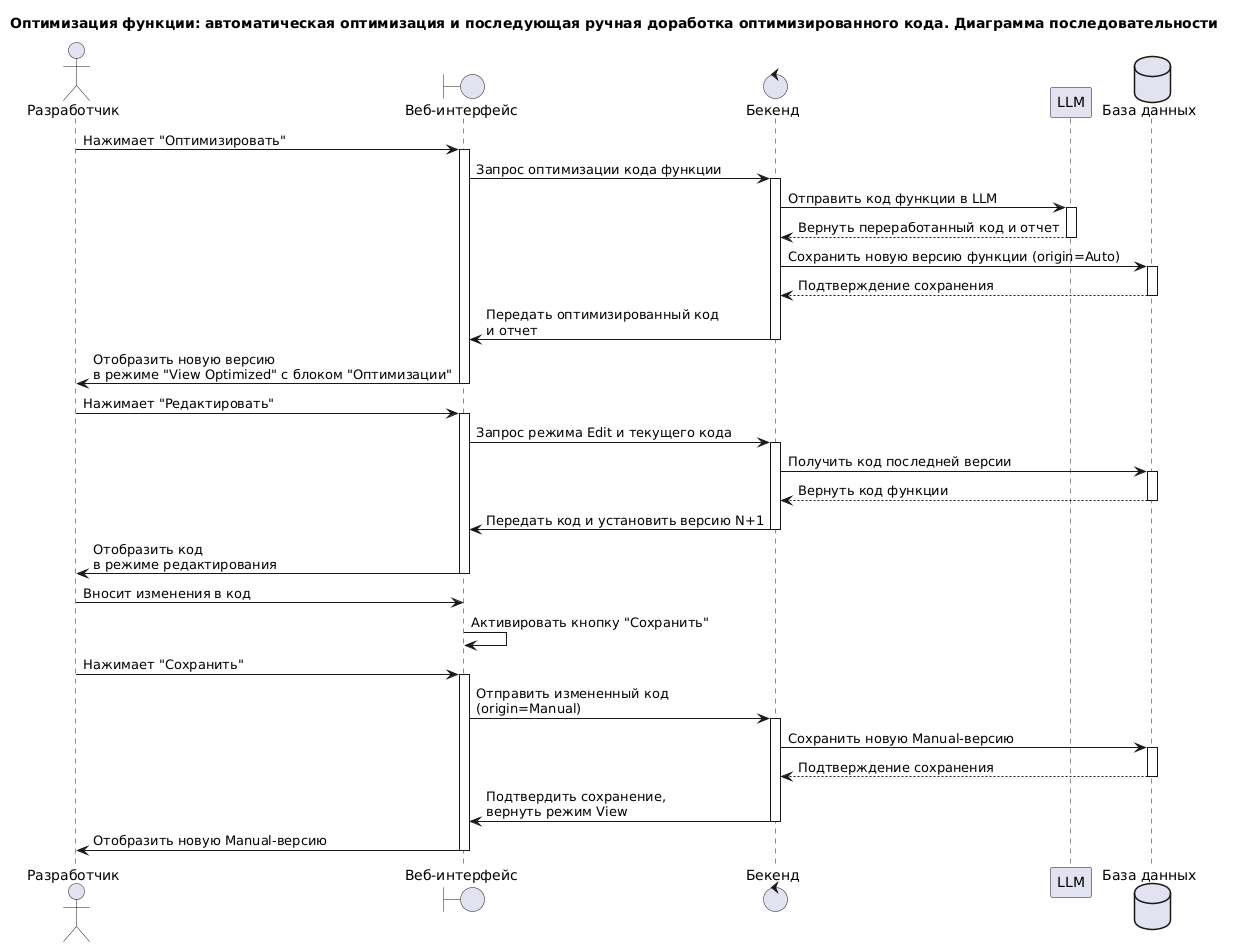


Рис. 2. Оптимизация функции: автоматическая оптимизация и последующая ручная доработка оптимизированного кода. Диаграмма последовательности

Особое внимание было уделено процессу тестирования функций с целью сбора объективных метрик производительности. Пользователю предоставляется возможность задавать параметры запускаемых тестов, такие как выбор конкретной версии функции, среду исполнения и временные ограничения выполнения задачи. По окончании тестирования система фиксирует метрики производительности, включая время запуска функции, продолжительность выполнения, использование центрального процессора и оперативной памяти. Данные показатели затем используются для анализа эффективности произведённых оптимизаций и принятия решений о выборе наилучшей конфигурации. Рисунок 3 содержит диаграмму последовательности данного процесса.

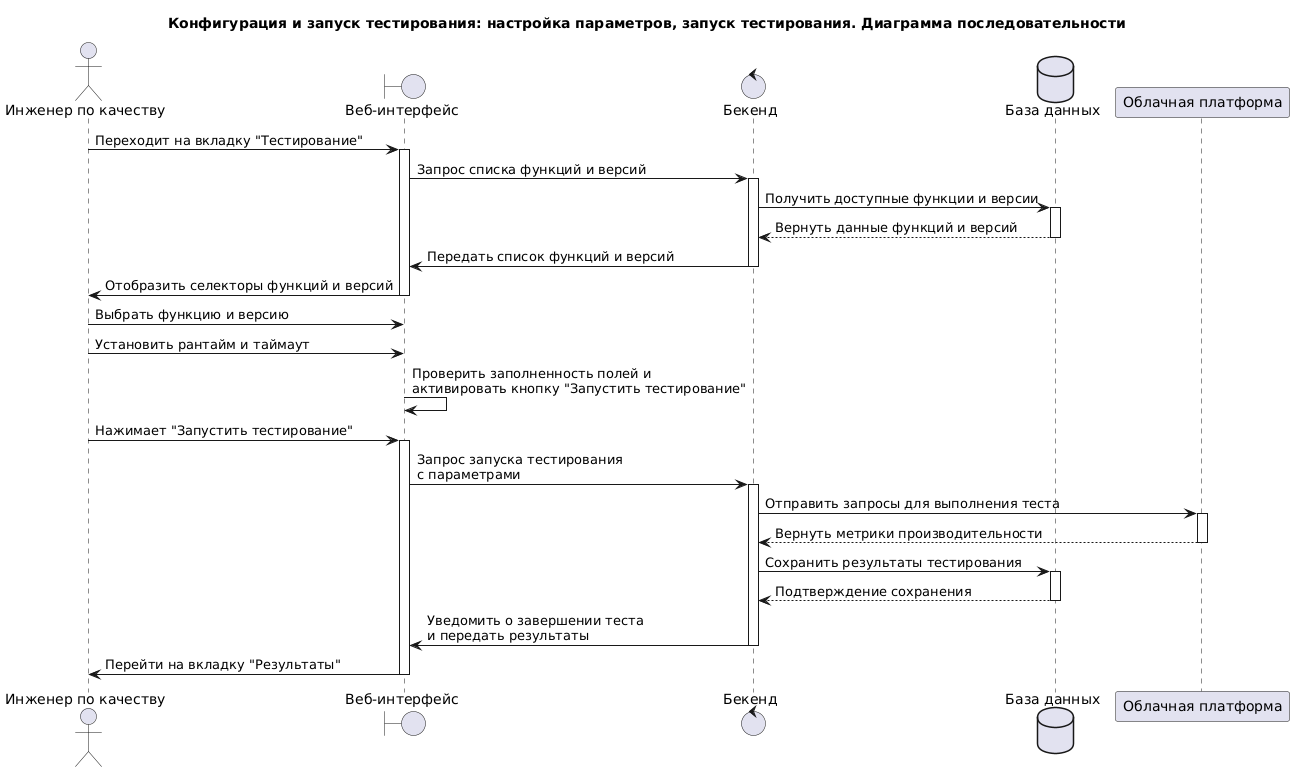


Рис. 3. Конфигурация и запуск тестирования: настройка параметров, запуск тестирования. Диаграмма последовательности

Для анализа и сравнения результатов работы функций в программном продукте предусмотрен специальный инструмент визуализации данных, представляющий собой веб-портал, где пользователи могут просматривать результаты тестирования отдельных версий функций и проводить их сравнение между собой. Пользователь получает возможность детально изучать графическое представление основных метрик, таких как время запуска, время выполнения, нагрузка на процессор и потребление памяти. При сравнении двух версий функций на одном графике отображаются оба набора данных, дополнительно формируется аналитическая сводка с указанием степени изменения ключевых параметров в процентах, обозначенных соответствующими цветовыми маркерами, которые визуально демонстрируют улучшение или ухудшение результатов тестирования. Диаграмма последовательности данного сценария представлена на рисунке 4.

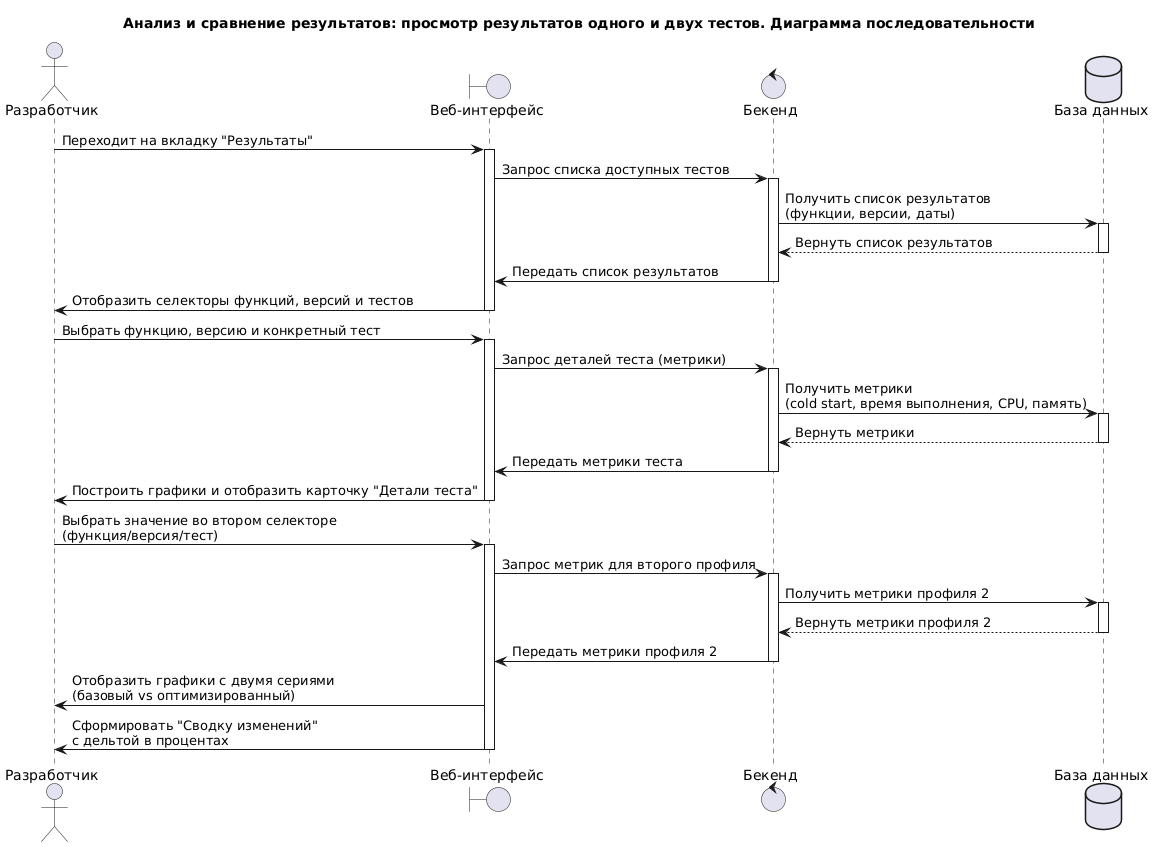


Рис. 4. Анализ и сравнение результатов: просмотр результатов одного и двух тестов. Диаграмма последовательности

Использование данного программного продукта предполагает ряд типовых сценариев взаимодействия, наиболее распространённым из которых является создание и первичное наполнение серверных функций программным кодом. В рамках данного сценария пользователь обращается к веб-интерфейсу продукта, после чего он заполняет необходимые данные и производит запись новой функции в систему. В результате выполнения таких действий создаётся исходная версия кода, доступная для последующего редактирования, автоматизированной оптимизации и тестирования.

Другим распространённым сценарием использования является процесс автоматической оптимизации функций, инициируемый пользователем посредством нажатия соответствующей кнопки в интерфейсе просмотра функции. Система, получив код, направляет его для анализа в подключённые большие языковые модели, после чего разработчику возвращается уже переработанная версия функции с перечнем изменений, направленных на повышение производительности. Новая версия автоматически сохраняется системой и становится доступной пользователю для последующего тестирования и ручной доработки.

Важную роль в процессе эксплуатации программного продукта играет возможность ручной корректировки как исходного, так и автоматически оптимизированного кода. В процессе взаимодействия с системой разработчик получает доступ к интерфейсу редактирования, который предоставляет возможность внести необходимые изменения, сохранив их в новой версии с соответствующей пометкой о ручном происхождении версии. Пользователь самостоятельно решает, какие изменения являются необходимыми и важными, и после выполнения правок сохраняет результаты работы для последующего анализа и использования в процессе тестирования.

Следующий важный сценарий использования подразумевает подготовку и запуск процесса тестирования функций. Разработчик или инженер по качеству выбирает версию функции, задаёт условия исполнения, включая параметры времени выполнения и ограничения по ресурсам, после чего запускает тестирование. Во время выполнения тестов система фиксирует метрики производительности, по завершении пользователю предоставляется возможность ознакомиться с итоговыми данными в специально предназначенном для этого разделе веб-интерфейса.

Заключительным этапом является анализ результатов тестирования и их визуальное представление для принятия решений. Пользователь, выбирая интересующую его версию функции, получает графическое представление основных метрик и возможность детально ознакомиться с подробностями выполнения каждого теста. Дополнительно доступна функция сравнения двух различных версий, результаты которой отображаются на одном графике, что даёт возможность объективно оценить эффективность проведённых оптимизаций и выбрать наиболее подходящие конфигурации функций.

## Выбор инструментов и технологий для разработки

При реализации серверной части программного комплекса предпочтение было отдано языку программирования Python версии 3.13. Выбор данной версии обусловлен доступностью новейших улучшений языка, включающих повышенную производительность, современные методы управления памятью и расширенные возможности асинхронного программирования. Python является одним из наиболее востребованных и широко используемых языков в области разработки серверных приложений, обеспечивающим богатый выбор библиотек и высокую скорость разработки [10].

Основным инструментом для реализации веб-интерфейса серверной части программного продукта выбран фреймворк FastAPI, который демонстрирует высокую производительность и удобство использования в асинхронной среде исполнения. FastAPI отличается строгой типизацией запросов и ответов, высокой скоростью обработки входящих данных и наличием встроенных механизмов документации API через спецификации OpenAPI. Благодаря этим особенностям FastAPI обеспечивает эффективную разработку надежного и прозрачного интерфейса взаимодействия с клиентскими приложениями, облегчая процессы тестирования и последующей поддержки системы.

Для обеспечения строгой валидации и типизации данных, поступающих и возвращаемых API, был выбран инструмент Pydantic версии 2. Данный инструмент позволяет эффективно контролировать соответствие данных заявленным типам и структурам, минимизируя вероятность ошибок, связанных с обработкой и передачей некорректной информации. Нововведения в Pydantic 2 включают значительные улучшения производительности, поддержку новых типов данных и глубокую интеграцию с современными фреймворками, такими как FastAPI. Для работы с данными, получаемыми от больших языковых моделей, была применена библиотека Pydantic AI, что позволило быстро интегрировать искусственный интеллект в серверный компонент разрабатываемого продукта.

Роль СУБД в приложении выполняет PostgreSQL, обладающая высокой производительностью и надежностью, а также поддержкой расширенного набора возможностей, включая работу с JSON-структурами и транзакционную безопасность. PostgreSQL хорошо показала себя в работе с большими объемами структурированных данных и активно применяется в сложных серверных приложениях, требующих высокой доступности и отказоустойчивости. Взаимодействие серверной части с базой данных реализовано посредством асинхронного драйвера AsyncPG, предоставляющего возможность выполнения неблокирующих операций ввода-вывода, что повышает пропускную способность и снижает время отклика системы в целом.

Для эффективной работы с базой данных на уровне приложения используется ORM-библиотека SQLAlchemy версии 2. Данная библиотека предоставляет удобный и прозрачный доступ к данным, а также абстрагирует разработчика от специфики конкретной базы данных, позволяя легко мигрировать на другие СУБД в случае необходимости. Новая версия SQLAlchemy обладает улучшенной архитектурой и поддержкой асинхронного программирования, что гармонично интегрируется с общей концепцией производительности и масштабируемости серверной части продукта.

В качестве технологии контейнеризации выбран Docker, обеспечивающий стабильную и воспроизводимую среду выполнения приложения. Использование контейнеризации позволяет разработчикам минимизировать проблемы совместимости и существенно упрощает развертывание системы на различных инфраструктурных платформах [11]. Docker предоставляет необходимые средства изоляции и управления ресурсами, что способствует упрощению развертывания и масштабирования разработанного приложения в облачной среде.

Для разработки клиентского приложения был выбран язык программирования JavaScript в паре с библиотекой React версии 18. React обеспечивает высокую производительность и гибкость, позволяя создавать масштабируемые и динамически изменяемые пользовательские интерфейсы. Современные подходы к управлению состоянием и компонентная архитектура React делают данный инструмент оптимальным решением для реализации интерактивных и сложных интерфейсов [12].

В качестве основной библиотеки компонентов пользовательского интерфейса выбрана Material-UI (MUI). Этот набор предлагает унифицированный и целостный UI-дизайн, основанный на концепциях Material Design от Google. Применение готовых компонентов позволяет существенно ускорить разработку и создать современный, визуально гармоничный и удобный интерфейс, соответствующий требованиям эргономики и пользовательского опыта.

Для маршрутизации в клиентском приложении был выбран инструмент React Router, предоставляющий удобный механизм управления навигацией и отображением различных экранов приложения без необходимости перезагрузки страницы. Это обеспечивает плавность взаимодействия пользователя с приложением и позволяет создавать динамичные интерфейсы, реагирующие на действия пользователя в режиме реального времени.

Взаимодействие с серверной частью осуществляется посредством HTTP-клиента Axios, который предоставляет удобный и простой интерфейс для выполнения запросов и обработки ответов от API. Axios отличается высокой надежностью и удобством работы с асинхронными запросами, предоставляя возможности перехвата и модификации запросов и ответов на различных этапах их жизненного цикла.

Для визуализации данных и представления результатов тестирования выбран инструмент Chart.js, обладающий широким спектром графических возможностей и простотой интеграции с React-приложениями. Chart.js позволяет отображать данные в различных форматах и графических формах, таких как линейные, столбчатые и круговые диаграммы, обеспечивая удобное и наглядное представление информации пользователю.

Сборка клиентского приложения и управление процессами разработки осуществляется посредством инструмента React Scripts, обеспечивающего автоматическую сборку, запуск локального сервера разработки и оптимизацию статических ресурсов приложения. React Scripts позволяет разработчикам быстро создавать и запускать проекты без необходимости настройки сложной инфраструктуры сборки.

Для автоматической генерации клиентского кода, соответствующего спецификациям API, используется инструмент openapi-typescript-codegen, который обеспечивает автоматизированную и согласованную работу с интерфейсами серверной части. Данный подход минимизирует возможность ошибок при взаимодействии с сервером и позволяет значительно ускорить процесс интеграции фронтенд- и бэкенд-компонентов продукта [13].

Аналогично серверной части, клиентское приложение также контейнеризируется с использованием технологии Docker, что обеспечивает одинаковую среду выполнения и упрощает процесс развертывания на целевой инфраструктуре. Использование контейнеризации способствует повышению надежности и предсказуемости работы приложения, а также облегчает совместную разработку и сопровождение программного комплекса в процессе его эксплуатации.

## Реализация фронтенда веб-приложения

Разработка пользовательского интерфейса программного комплекса осуществлялась с применением фреймворка React. Структура фронтенд-части приложения организована с целью обеспечения модульности, переиспользуемости компонентов и удобства дальнейшей поддержки и масштабирования. Корнем фронтенда является директория «frontend». Внутри данной директории располагаются стандартные для React-проектов конфигурационные файлы, такие как «package.json», определяющий зависимости проекта и скрипты сборки, и «tsconfig.json» для настройки TypeScript. Директория «public» содержит статические файлы, доступные напрямую через веб-сервер, например, «index.html» — основную HTML-страницу, в которую встраивается React-приложение. Дерево файлов приложения отражено в листинге 1.

Листинг 1. Структура приложения

$ tree -L 3 -I node\_modules ./frontend

./frontend

├── docker-compose.yml

├── Dockerfile

├── Dockerfile.dev

├── openapitools.json

├── package-lock.json

├── package.json

├── public

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── serverless-icon.svg

├── README.md

├── src

│ ├── api-client

│ │ ├── core

│ │ ├── index.ts

│ │ ├── models

│ │ └── services

│ ├── App.js

│ ├── components

│ │ └── Layout.js

│ ├── index.css

│ ├── index.js

│ ├── pages

│ │ ├── FunctionsPage.js

│ │ ├── OptimizationPage.js

│ │ ├── ResultsPage.js

│ │ └── TestingPage.js

│ └── reportWebVitals.js

└── tsconfig.json

Основная логика и компоненты пользовательского интерфейса сосредоточены в директории «src». Входной точкой приложения выступает файл «src/index.js». Данный файл отвечает за инициализацию React-приложения и его монтирование в DOM-дерево HTML-страницы. В «src/index.js» выполняется подключение основного компонента приложения, «App», а также настройка глобальных стилей из «src/index.css» и интеграция системы маршрутизации «react-router-dom» посредством компонента «BrowserRouter». Полное содержание файла «src/index.js» продемонстрировано на рисунке 5.

A screen shot of a computer code

AI-generated content may be incorrect.

Рис. 5. Содержимое файла «src/index.js»

Центральным компонентом, агрегирующим основные элементы интерфейса и логику маршрутизации, является «src/App.js». В данном компоненте определяется тема оформления приложения с использованием библиотеки Material-UI, обеспечивающей консистентный внешний вид элементов управления. «App.js» также содержит в себе компонент «Layout» и определяет основные маршруты приложения с помощью «Routes» и «Route» из «react-router-dom». Маршруты связывают URL-адреса с соответствующими компонентами страниц, такими как «FunctionsPage», «OptimizationPage», «TestingPage» и «ResultsPage». Полностью содержимое файла «src/App.js» приведено в Приложении №1.

Структура проекта предполагает разделение компонентов на две основные категории: переиспользуемые UI-элементы, расположенные в директории «src/components», и компоненты страниц, находящиеся в «src/pages». На данный момент в «src/components» присутствует единственный компонент «Layout.js». Этот компонент определяет общую структуру веб-страниц приложения, включая навигационное меню и область для отображения основного контента. «Layout.js» использует компоненты из Material-UI, такие как «AppBar», «Drawer», «List», «ListItem» для создания адаптивного интерфейса, корректно отображающегося на различных размерах экранов. Навигация между разделами приложения осуществляется с помощью функций «useNavigate» и «useLocation» из «react-router-dom». Код файла «src/components/Layout.js» расположен в Приложении №2

Директория «src/pages» содержит компоненты, представляющие отдельные страницы приложения: «FunctionsPage.js» для работы со списком функций, «OptimizationPage.js» для взаимодействия с LLM и получения оптимизированных версий кода, «TestingPage.js» для запуска тестов производительности и «ResultsPage.js» для визуализации результатов тестирования. Каждый из этих файлов реализует логику и пользовательский интерфейс соответствующего раздела. Каждая страница, в свою очередь, имеет собственные состояния, переход между которыми вызывается нажатием кнопок, взаимодействием пользователя с различными элементами интерфейса, или изменением состояния запроса. Состояния станиц и переходы между ними отражены на диаграмме состояний (рисунок 6).

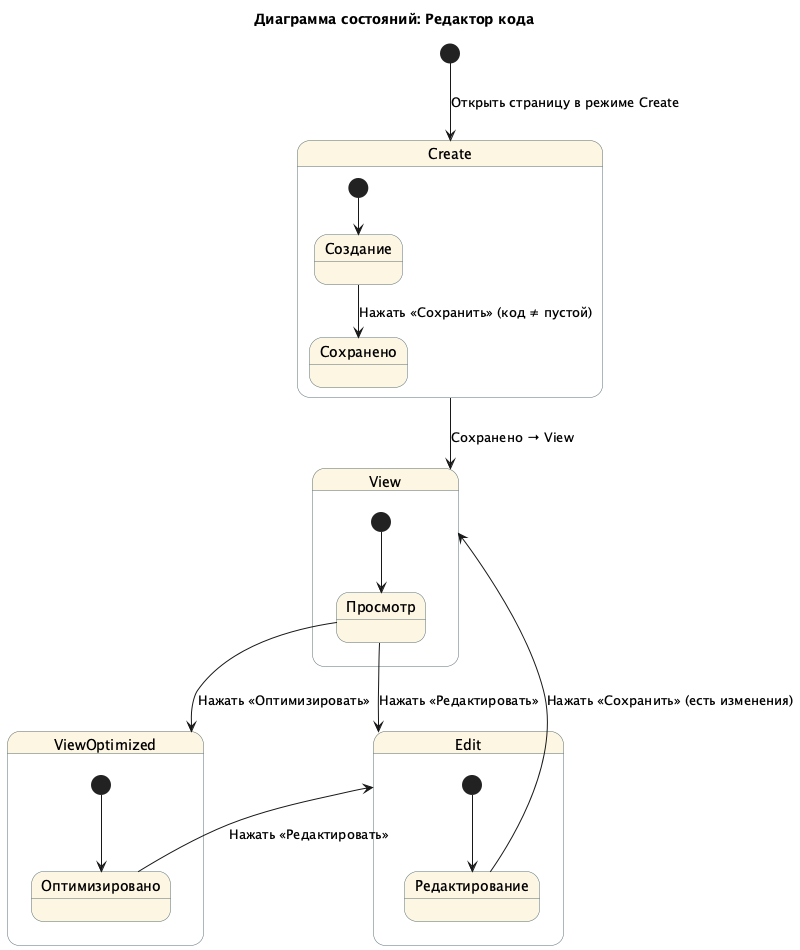


Рис. 6. Диаграмма состояний приложения

Взаимодействие с бекенд-частью приложения осуществляется через автоматически сгенерированный API-клиент, расположенный в директории «src/api-client». Генерация клиента производится на основе спецификации OpenAPI, что обеспечивает строгое соответствие между фронтендом и бекендом в части форматов запросов и ответов. Директория «src/api-client» включает поддиректории «core» для базовой логики клиента, «models» для описания моделей данных и «services» для сервисов, вызывающих методы API. Файл «src/api-client/index.ts» экспортирует основные сущности API-клиента для использования в других частях приложения.

В ходе разработки особое внимание уделялось переиспользуемости компонентов. Элементы интерфейса, которые могли бы повторяться на нескольких страницах, выносились в отдельные компоненты в директорию «src/components». Такой подход способствовал уменьшению дублирования кода и упрощению его поддержки. Стилизация компонентов производилась с использованием встроенных возможностей Material-UI, а также через CSS-модули или глобальные стили в «src/index.css» для общих правил оформления. Решения о структуре данных и потоках информации внутри приложения принимались итерационно, по мере реализации функциональности каждой страницы и компонента. Использование системы контроля версий Git позволило отслеживать изменения и координировать работу [14]. Дальнейшее развитие фронтенд-части предполагает расширение набора компонентов, улучшение пользовательского опыта и добавление новой функциональности в соответствии с требованиями к программному комплексу. Скриншот итоговой домашней страницы приложения находится на рисунке 7.

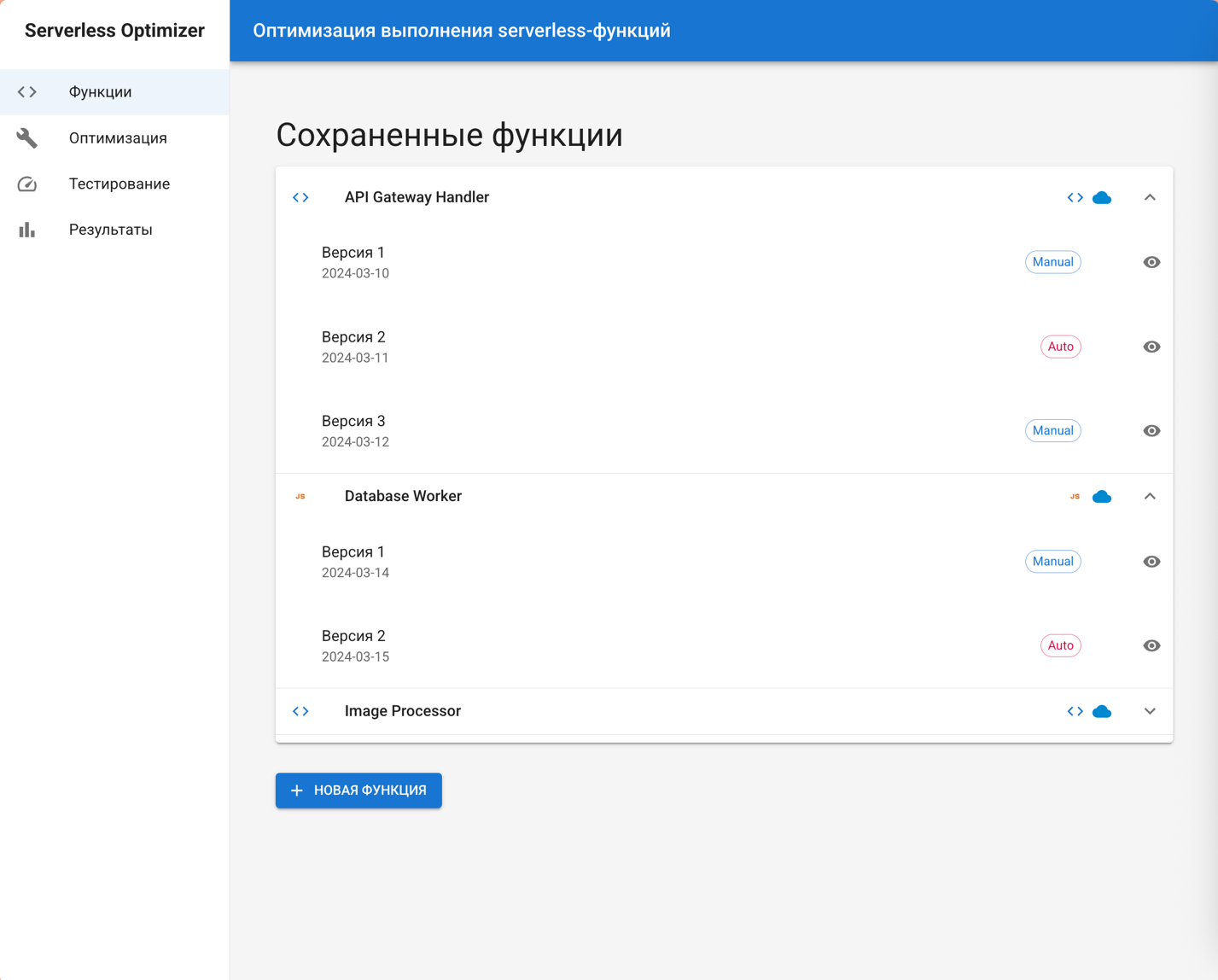


Рис. 7. Скриншот домашей страницы веб-интерфейса приложения

## Реализация бекенда веб-приложения

Разработка серверной части комплекса осуществлялась с применением языка программирования Python и веб-фреймворка FastAPI. Выбор данных технологий обусловлен их производительностью, асинхронной природой, что критически важно для взаимодействия с облачными сервисами и моделями машинного обучения, а также обширной экосистемой библиотек. Процесс разработки начался с определения ключевых сущностей системы и их взаимосвязей, что легло в основу модульной архитектуры приложения. Архитектура бекенда программного комплекса верхнеуровнево представлена на рисунке 8.

A diagram of a company

AI-generated content may be incorrect.

Рис. 8. Архитектура бекенда программного комплекса

Структура проекта организована таким образом, чтобы обеспечить четкое разделение ответственности между компонентами. Корневая директория серверной части, именуемая «backend», содержит конфигурационные файлы, такие как «pyproject.toml» для управления зависимостями с помощью инструмента «uv», и «Dockerfile» для контейнеризации приложения. Основная логика приложения сосредоточена в поддиректории «app». Дерево файлов приложения приведено в листинге 2.

Листинг 2. Структура приложения

$ tree -I ".venv|\_\_pycache\_\_" ./backend

./backend

├── app

│ ├── \_\_init\_\_.py

│ ├── core

│ │ ├── \_\_init\_\_.py

│ │ ├── cloud\_platform.py

│ │ ├── config.py

│ │ ├── db.py

│ │ ├── llm.py

│ │ ├── schemas.py

│ │ └── testing.py

│ ├── main.py

│ ├── models.py

│ ├── routers

│ │ ├── \_\_init\_\_.py

│ │ ├── dependencies.py

│ │ ├── functions.py

│ │ ├── optimization.py

│ │ ├── reference.py

│ │ ├── testing.py

│ │ └── versions.py

│ └── services

│ ├── \_\_init\_\_.py

│ └── optimization\_service.py

├── docker-compose.yml

├── Dockerfile

├── pyproject.toml

└── uv.lock

Внутри «app» выделяются несколько ключевых модулей. Файл «main.py» выступает в качестве точки входа в приложение. В нем инициализируется экземпляр «FastAPI», настраиваются политики CORS для междоменных запросов и подключаются роутеры из пакета «routers». Также в «main.py» определяется асинхронный менеджер контекста «lifespan», который отвечает за инициализацию соединения с базой данных при старте приложения и его корректное закрытие при остановке. Инициализация приложения FastAPI в файле «main.py» продемонстрирована в листинге 3.

Листинг 3. Фрагмент файла «main.py»

@asynccontextmanager

async def lifespan(app: FastAPI) -> AsyncGenerator[None, None]:

await init\_db()

yield

app = FastAPI(

title="Serverless Function Optimization API",

description="REST API for automated execution, measurement and optimization of "

"serverless functions on import-substitution cloud infrastructure.",

version="1.0.0",

lifespan=lifespan,

)

app.include\_router(functions.router)

app.include\_router(versions.router)

app.include\_router(optimization.router)

app.include\_router(testing.router)

app.include\_router(reference.router)

Модели данных, используемые для валидации входящих запросов и формирования ответов, определены в файле «models.py». Здесь с помощью библиотеки «Pydantic» описаны различные классы, представляющие сущности системы, такие как «FunctionCreateRequest», «TestRunResponse», «LLMProviderType» и другие. Эти модели обеспечивают строгую типизацию данных и автоматическую валидацию, что снижает вероятность ошибок при обмене информацией между клиентом и сервером. Листинг 4 содержит примеры объявления моделей.

Листинг 4. Фрагмент файла «models.py»

from datetime import datetime

from enum import Enum

from typing import Any

from uuid import UUID

from pydantic import BaseModel, Field

class Language(str, Enum):

NODE\_JS = "node\_js"

PYTHON = "python"

class Platform(str, Enum):

YANDEX\_CLOUD\_FUNCTIONS = "yandex\_cloud\_functions"

class FunctionSummary(BaseModel):

id: int

name: str

language: Language

platform: Platform

versions: list[VersionSummary]

class FunctionListResponse(BaseModel):

functions: list[FunctionSummary]

class FunctionCreateRequest(BaseModel):

name: str

language: Language

platform: Platform

source\_code: str

class FunctionCreateResponse(BaseModel):

function\_id: int

version: int

Директория «core» содержит основную бизнес-логику приложения. В модуле «config.py» определен класс «Settings», использующий возможности библиотеки «pydantic-settings» для чтения настроек приложения из переменных окружения и файла «.env». Эти настройки включают в себя URI для подключения к базе данных, API-токены облачных платформ и больших языковых моделей, а также другие настройки. Файл «db.py» определяет асинхронные коннекторы для PostgreSQL, используя для этого библиотеку «SQLAlchemy». Он хранит базовый класс для декларативных моделей «Base», создает асинхронный движок «engine» и фабрику сессий «async\_session\_factory». Функция «init\_db» вызывается при старте приложения для создания таблиц в базе данных на основе определенных моделей при запуске приложения. Код подключения к БД продемонстрирован в листинге 5.

Листинг 5. Настройка соединения с базой данных в db.py

engine = create\_async\_engine(str(settings.POSTGRES\_DSN))

async\_session\_factory = async\_sessionmaker(

engine, expire\_on\_commit=False, autoflush=False

)

async def get\_db() -> AsyncGenerator[AsyncSession, None]:

async with async\_session\_factory() as session:

try:

yield session

finally:

await session.close()

Схемы таблиц базы данных определены в файле «schemas.py». Здесь с применением SQLAlchemy ORM описаны такие сущности, как «Function», «Version», «OptimizationTask», «TestRun», «Profile», «Comparison» и «YCFVersion». Эти классы отображают структуру таблиц базы данных и определяют связи между ними, например, одна функция может иметь множество версий, а каждая версия может быть связана с несколькими задачами оптимизации или тестовыми запусками. Пример определения ORM-модели таблицы приведен в Приложении №3.

Взаимодействие с облачной платформой, в данном случае «Yandex Cloud Functions», инкапсулировано в классе «YandexCloudExecutor» в модуле «cloud\_platform.py». Этот класс отвечает за создание пакетов с кодом функций, их загрузку в облако, создание версий функций и их вызов. Решение инкапсулировать логику взаимодействия с конкретной облачной платформой в этом классе было принято для упрощения возможного добавления поддержки других провайдеров в будущем. Логика тестирования функций, включая запуск кода, сбор результатов и обработку ошибок, реализована в классе «TestRunner» в модуле «testing.py». «TestRunner» использует экземпляр «YandexCloudExecutor» для выполнения функций в облаке и сохраняет результаты в базу данных.

Интеграция с большими языковыми моделями для оптимизации кода функций реализована в модуле «llm.py». Он содержит функции для формирования промптов, взаимодействия с API моделей «OpenAI» или «Anthropic» с помощью библиотеки «Pydantic-AI», и обработки результатов. Функция «optimize\_function» принимает на вход исходный код, язык, платформу и провайдера LLM, а возвращает оптимизированный код и список примененных оптимизаций. Решение об использовании Pydantic-AI было принято для унификации работы с различными моделями и упрощения парсинга структурированных ответов. Код для взаимодействия с большой языковой моделью приведен в листинге 6.

Листинг 6. Фрагмент кода для взаимодействия с LLM из llm.py

async def optimize\_function(input\_data: OptimizationInput) -> OptimizationOutput:

agent = get\_agent(input\_data.provider)

user\_prompt = get\_optimization\_prompt(input\_data)

result = await agent.run(

user\_prompt,

output\_type=LLMOptimizationResult,

message\_history=[],

)

llm\_result = result.output

return OptimizationOutput(

optimized\_code=llm\_result.optimized\_code,

language=input\_data.language,

platform=input\_data.platform,

optimizations=llm\_result.optimizations,

recommended\_runtime=llm\_result.recommended\_runtime,

provider=input\_data.provider,

)

Маршрутизация HTTP-запросов осуществляется в модулях, расположенных в директории «routers». Каждый модуль в этом пакете, например, «functions.py», «versions.py», «optimization.py», «testing.py» и «reference.py», определяет набор эндпоинтов, связанных с определенной группой сущностей. Например, «functions.py» обрабатывает запросы на создание, получение списка и удаление функций. Файл «dependencies.py» содержит общие зависимости, используемые в различных обработчиках в рамках паттерна проектирования «dependency injection», используемые, например, для получения объекта функции или версии функции по идентификатору из базы данных. Использование механизма внедрения зависимостей «FastAPI» позволяет вынести повторяющуюся логику из обработчиков запросов, делая код более чистым и поддерживаемым [15].

Особенностью веб-фреймворка FastAPI является автоматическая генерация документации к API приложения в формате OpenAPI. Спецификация OpenAPI также применяется для автоматической генерации интерактивной документации приложения с помощью инструментов «Swagger» и «ReDoc». Swagger-документация приложения представлена на рисунке 9.

A screenshot of a computer

AI-generated content may be incorrect.

Рис. 9. Скриншот интерактивной документации API

Сервисный слой приложения расположен в директории «services». В частности, файл «optimization\_service.py» содержит логику управления задачами оптимизации. Функция «start\_optimization\_task» создает новую задачу оптимизации в базе данных со статусом «QUEUED» и запускает асинхронную фоновую задачу «process\_optimization\_task» для ее выполнения. Эта фоновая задача изменяет статус на «RUNNING», вызывает модуль «llm.py» для получения оптимизированного кода, создает новую версию функции с этим кодом и обновляет статус задачи на «COMPLETED» или «FAILED» в случае ошибки. Такое разделение на сервисы позволяет инкапсулировать бизнес-логику и сделать ее независимой от конкретных деталей реализации API.

Изначально рассматривался вариант монолитной структуры без явного выделения сервисного слоя, однако по мере усложнения логики оптимизации и тестирования было принято решение о его внедрении для улучшения читаемости и тестируемости кода. Также на ранних этапах разработки предполагалось использование единой модели для представления функции и ее версии, но для более гибкого управления исходным кодом и результатами оптимизаций было решено разделить эти сущности. Выбор асинхронного подхода с самого начала проекта позволил избежать необходимости внесения значительных изменений при добавлении взаимодействия с внешними API, которые по своей природе являются блокирующими операциями. Постоянное внимание к типизации и использование возможностей Pydantic и SQLAlchemy способствовало созданию надежной и предсказуемой системы.

# ЗАКЛЮЧЕНИЕ

В ходе работы был изучен комплекс вопросов, связанных с автоматизацией оптимизации и оценки производительности serverless-функций, размещенных на национальной облачной инфраструктуре. Были рассмотрены подходы к оптимизации serverless-технологий, а также ключевые метрики производительности, среди которых время выполнения, потребление ресурсов ЦП и время холодного старта. Также были выявлены отличительные особенности работы с отечественной платформой Yandex Cloud Functions в сравнении с зарубежным аналогом AWS Lambda, что позволило более точно определить специфику и требования к разрабатываемому продукту.

Практическая часть работы заключалась в разработке и реализации программного комплекса, включающего серверную часть и веб-интерфейс, позволяющего автоматически запускать и оптимизировать serverless-функции. Были успешно подобраны и внедрены современные технологии, среди которых FastAPI, Python, PostgreSQL и инструменты асинхронного программирования, а также React, JavaScript и Material-UI для реализации интерфейсной части. В ходе разработки был реализован модуль интеграции с большими языковыми моделями, позволяющий автоматически генерировать рекомендации по оптимизации кода и создавать улучшенные версии функций.

Разработанный веб-интерфейс предоставляет пользователю возможности создания, автоматической оптимизации и ручного редактирования функций, а также настройки и запуска тестирования с последующим наглядным отображением результатов, и метрик производительности. Визуализация данных позволяет объективно оценить эффективность произведенных изменений и выбрать наилучшую конфигурацию serverless-функций для дальнейшей эксплуатации.

Все задачи, поставленные во введении, были успешно выполнены, что подтверждает достижение основной цели работы — разработки программного комплекса для автоматизированного запуска, анализа и оптимизации производительности serverless-функций. Предполагаемый результат работы был полностью реализован, а созданный продукт имеет высокую практическую ценность. Разработанный комплекс позволит организациям и государственным учреждениям эффективно решать задачи импортозамещения и повысить общую эффективность эксплуатации своих информационных систем на национальной облачной платформе. В дальнейшем возможны расширение функционала, интеграция новых моделей анализа и оптимизации, а также поддержка дополнительных облачных платформ. Выполненная работа демонстрирует высокую актуальность и востребованность разработанного решения для российских предприятий и государственных учреждений.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Щербинина Мария Юрьевна, Крюкова Анастасия Александровна Импортозамещение в ИТ-сфере // КНЖ. 2016. №4 (17). URL: https://cyberleninka.ru/article/n/importozameschenie-v-it-sfere (дата обращения: 04.06.2025).
2. Макосий Алексей Иванович, Макосий Роман Современная облачная инфраструктура: бессерверные вычисления // Вестник ХГУ им. Н. Ф. Катанова. 2019. №28. URL: https://cyberleninka.ru/article/n/sovremennaya-oblachnaya-infrastruktura-besservernye-vychisleniya (дата обращения: 04.06.2025).
3. Amazon Web Services. AWS Lambda [Электронный ресурс]. Режим доступа: https://aws.amazon.com/lambda/. Дата обращения: 04.06.2025.
4. Маркаданов А. А. ВОЗМОЖНОСТИ CHATGPT В СФЕРЕ ПРОГРАММИРОВАНИЯ // Вестник науки. 2025. №2 (83). URL: https://cyberleninka.ru/article/n/vozmozhnosti-chatgpt-v-sfere-programmirovaniya (дата обращения: 04.06.2025).
5. Гордина А. Т., Забродин А. В. Особенности технологий бессерверных вычислений // Интеллектуальные технологии на транспорте. 2022. №1 (29). URL: https://cyberleninka.ru/article/n/osobennosti-tehnologiy-besservernyh-vychisleniy (дата обращения: 04.06.2025).
6. Yandex.Cloud [Электронный ресурс]. Режим доступа: https://yandex.cloud/ru/. Дата обращения: 04.06.2025.
7. Selectel. Руководство по запуску Serverless-приложений [Электронный ресурс]. Режим доступа: https://selectel.ru/blog/serverless-guide/. Дата обращения: 04.06.2025.
8. Cloud.ru. FunctionGraph Serverless [Электронный ресурс]. Режим доступа: https://cloud.ru/products/functiongraph-serverless. Дата обращения: 04.06.2025.
9. Yandex.Cloud. Сопоставление с Amazon Web Services [Электронный ресурс]. Режим доступа: https://yandex.cloud/ru/docs/overview/platform-comparison/aws. Дата обращения: 04.06.2025.
10. Шершень Кирилл Викторович ВОСТРЕБОВАННЫЕ BACK-END-ТЕХНОЛОГИИ ДЛЯ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА // Universum: технические науки. 2024. №1 (118). URL: https://cyberleninka.ru/article/n/vostrebovannye-back-end-tehnologii-dlya-razrabotki-programmnogo-produkta (дата обращения: 04.06.2025).
11. Чиганов Д. Р. DOCKER: КЛЮЧ К КОНТЕЙНЕРИЗАЦИИ И МАСШТАБИРУЕМОСТИ // Вестник науки. 2023. №7 (64). URL: https://cyberleninka.ru/article/n/docker-klyuch-k-konteynerizatsii-i-masshtabiruemosti (дата обращения: 04.06.2025).
12. Мамбетов Р. А. ПОСТРОЕНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ ВЕБ-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ БИБЛИОТЕКИ REACT // Экономика и социум. 2019. №5 (60). URL: https://cyberleninka.ru/article/n/postroenie-polzovatelskih-interfeysov-veb-prilozheniy-s-pomoschyu-biblioteki-react (дата обращения: 04.06.2025).
13. Волушкова Вера Львовна, Волушкова Александра Юрьевна ЕДИНЫЙ ФОРМАТ СПЕЦИФИКАЦИИ В КАЧЕСТВЕ API-АРТЕФАКТА МИКРОСЕРВИСА ПРИ ИСПОЛЬЗОВАНИИ API-FIRST // Программные системы и вычислительные методы. 2022. №4. URL: https://cyberleninka.ru/article/n/edinyy-format-spetsifikatsii-v-kachestve-api-artefakta-mikroservisa-pri-ispolzovanii-api-first (дата обращения: 04.06.2025).
14. Хомоненко Анатолий Дмитриевич, Каратаев Евгений Николаевич Использование продвинутых функций Git при разработке программного обеспечения // Интеллектуальные технологии на транспорте. 2024. №2 (38). URL: https://cyberleninka.ru/article/n/ispolzovanie-prodvinutyh-funktsiy-git-pri-razrabotke-programmnogo-obespecheniya (дата обращения: 04.06.2025).
15. Уваров Артем Николаевич Инверсия управления и внедрение зависимостей // Символ науки. 2016. №10-1. URL: https://cyberleninka.ru/article/n/inversiya-upravleniya-i-vnedrenie-zavisimostey (дата обращения: 04.06.2025).

# ПРИЛОЖЕНИЯ

Приложение №1. Содержимое файла «src/App.js»

import React from 'react';

import { Routes, Route } from 'react-router-dom';

import { ThemeProvider, createTheme } from '@mui/material/styles';

import CssBaseline from '@mui/material/CssBaseline';

import Layout from './components/Layout';

import FunctionsPage from './pages/FunctionsPage';

import OptimizationPage from './pages/OptimizationPage';

import TestingPage from './pages/TestingPage';

import ResultsPage from './pages/ResultsPage';

const theme = createTheme({

palette: {

primary: {

main: '#1976d2',

},

secondary: {

main: '#dc004e',

},

background: {

default: '#f5f5f5',

},

},

typography: {

fontFamily: 'Roboto, Arial, sans-serif',

},

});

function App() {

return (

<ThemeProvider theme={theme}>

<CssBaseline />

<Layout>

<Routes>

<Route path="/" element={<FunctionsPage />} />

<Route path="/optimization" element={<OptimizationPage />} />

<Route path="/optimization/:functionId" element={<OptimizationPage />} />

<Route path="/optimization/:functionId/:version" element={<OptimizationPage />} />

<Route path="/testing" element={<TestingPage />} />

<Route path="/testing/:functionId/:version" element={<TestingPage />} />

<Route path="/results" element={<ResultsPage />} />

</Routes>

</Layout>

</ThemeProvider>

);

}

export default App;

Приложение №2. Фрагмент файла «src/components/Layout.js»

import React, { useState } from 'react';

import { useNavigate, useLocation } from 'react-router-dom';

import {

AppBar,

Box,

Drawer,

IconButton,

List,

ListItem,

ListItemIcon,

ListItemText,

Toolbar,

Typography,

useMediaQuery,

useTheme,

} from '@mui/material';

import MenuIcon from '@mui/icons-material/Menu';

import FunctionsIcon from '@mui/icons-material/Code';

import BuildIcon from '@mui/icons-material/Build';

import SpeedIcon from '@mui/icons-material/Speed';

import BarChartIcon from '@mui/icons-material/BarChart';

const drawerWidth = 240;

const menuItems = [

{ text: 'Функции', icon: <FunctionsIcon />, path: '/' },

{ text: 'Оптимизация', icon: <BuildIcon />, path: '/optimization' },

{ text: 'Тестирование', icon: <SpeedIcon />, path: '/testing' },

{ text: 'Результаты', icon: <BarChartIcon />, path: '/results' },

];

function Layout({ children }) {

const theme = useTheme();

const isMobile = useMediaQuery(theme.breakpoints.down('md'));

const [mobileOpen, setMobileOpen] = useState(false);

const navigate = useNavigate();

const location = useLocation();

const handleDrawerToggle = () => {

setMobileOpen(!mobileOpen);

};

// ... остальной код компонента

return (

<Box sx={{ display: 'flex' }}>

{/\* ... код AppBar и Drawer ... \*/}

<Box

component="main"

sx={{

flexGrow: 1,

p: 3,

width: { sm: `calc(100% - ${drawerWidth}px)` },

}}

>

<Toolbar />

{children}

</Box>

</Box>

);

}

export default Layout;

Приложение №3. Пример определения ORM-модели таблицы

class Function(Base):

\_\_tablename\_\_ = "functions"

id: Mapped[int] = mapped\_column(Integer, primary\_key=True, autoincrement=True)

name: Mapped[str] = mapped\_column(String(255), nullable=False)

language: Mapped[Language] = mapped\_column(Enum(Language), nullable=False)

platform: Mapped[Platform] = mapped\_column(Enum(Platform), nullable=False)

ycf\_id: Mapped[str | None] = mapped\_column(String(255), nullable=True, unique=True)

versions: Mapped[list["Version"]] = relationship(

back\_populates="function",

cascade="all, delete-orphan",

order\_by="Version.version\_number",

)