

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Система мониторинга символьного исполнения для KLEEF

Кононова Юлия Алексеевна

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
к. ф.-м. н Д.А. Мордвинов

Консультант:
инженер-исследователь кафедры системного программирования М.П. Костицын

Санкт-Петербург
2024

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
2.1. Символьное исполнение	5
2.2. Обзор требований к реализуемому приложению	6
2.3. Обзор существующих решений	7
2.4. Обзор технологий для реализации приложения	8
3. Архитектура	11
3.1. Архитектура серверной части	11
3.2. Архитектура клиентской части	12
3.3. Клиент-серверное взаимодействие	13
3.4. Взаимодействие с агентом	14
4. Детали реализации	16
4.1. Формат метрик	16
4.2. JacoDB и RAMEntityRelationshipStorage	16
5. Тестирование	18
Заключение	19
Список литературы	20

Введение

В современном мире, где информационные системы становятся все более сложными и масштабными, мониторинг их работы приобретает критическую важность. Большие системы, состоящие из множества взаимодействующих компонентов, требуют постоянного наблюдения и контроля для обеспечения стабильности, производительности и безопасности.

Среди таких систем выделяется система анализа кода, так как в ходе ее работы может возникать потребность в сборе различных метрик, характеризующих ее состояние в разные моменты времени. В дальнейшем полученные данные могут быть использованы как для отладки самой системы, так и для повышения качества анализа.

Одной из техник, активно применяемых для анализа программного обеспечения, является символьное исполнение. Эта техника заключается в исполнении программы не на конкретных входных данных, а на символьных переменных. Это позволяет исследовать все возможные пути исполнения программы. Одним из применений символьного исполнения является автоматическая генерация тестов. Ярким примером инструмента, использующего технику символьного исполнения для генерации тестового покрытия, является KLEEF.

KLEEF[7] — это инструмент, разработанный на основе механизма символьного исполнения KLEE[6], который оптимизирован для анализа промышленного кода на языках C и C++, а также генерации тестового покрытия.

На данный момент процесс символьного исполнения остается сложным и ресурсоемким. Для обеспечения эффективности и контроля над этим процессом необходимо создать систему мониторинга, которая позволит отслеживать работу KLEEF, выявлять потенциальные проблемы и узкие места, а также предоставлять аналитическую информацию для дальнейшей оптимизации. В рамках данной работы планируется разработать систему мониторинга символьного исполнения для KLEEF.

1 Постановка задачи

Цель работы — создать веб-приложение, направленное на мониторинг символьного исполнения для инструмента KLEEF. Для достижения цели были поставлены следующие задачи:

1. выполнить обзор технологий символьного исполнения и технологий для создания веб-приложения, реализующих функциональность мониторинг системы;
2. сформулировать требования;
3. разработать архитектуру приложения на основе выявленных требований;
4. реализовать приложение согласно архитектуре;
5. протестировать реализованное приложение.

2 Обзор

В данном разделе будут рассмотрены: предметная область, существующие решения и инструмент KLEEF, для которого будет реализована мониторинг система. Также будут выработаны требования к разрабатываемому приложению. На их основе будет выполнен обзор основных технологий для ее реализации.

2.1 Символьное исполнение

Символьное исполнение[14] - это метод анализа, применяемый для верификации и тестирования программного обеспечения. Метод основывается на интерпретации исследуемой программы с неопределенными входными значениями. Это означает, что все передаваемые аргументы заменяются на символьные константы и операции проводятся уже над ними. Такой подход позволяет сохранять ограничения на символьные константы, возникающие по ходу интерпретации. В качестве результата получается множество состояний, каждое из которых может принимать анализируемая программа, если входные аргументы удовлетворяют ограничениям соответствующих символьных констант. Для проверки достижимости конкретного состояния используется SMT-решатель. SMT (satisfiability modulo theories) - задача разрешимости для логических формул, учитывающая лежащие в основе теории.

При интерпретации программного обеспечения требуется контролировать ход исполнения. Мониторинг необходим для отслеживания состояния программы на каждом шаге и для своевременного обнаружения ошибок и исключений.

2.1.1 KLEEF

KLEE [6] — это мощный инструмент символьного исполнения, разработанный для автоматического тестирования программ на C и C++. Он позволяет анализировать код с целью нахождения ошибок, генерируя набор тестов, которые покрывают различные пути исполнения

программы.

KLEEF [7] — модифицированная версия KLEE. Основной целью KLEEF является улучшение производительности и масштабируемости символьного исполнения. Это достигается за счет улучшения алгоритмов поиска и оптимизации работы с ограничениями, что позволяет существенно сократить время выполнения и количество ресурсов, необходимых для анализа сложных программных систем.

Кроме того KLEEF имеет настроенные режимы как для генерации максимального тестового покрытия, так и для воспроизведения трассировок ошибок, в частности, для достижения определенной точки в программе. Также, согласно статье [4], KLEEF имеет поддержку SMT-решателя Bitwuzla [11], которая значительно лучше работает на тестах Test-Comp 2023[14].

2.2 Обзор требований к реализуемому приложению

Для разработки клиент-серверного веб-приложения, с четко разделенными обязанностями между серверной и клиентской частями, необходимо учесть ряд функциональных требований:

- сервер должен иметь возможность принимать JSON-запросы от внешних источников. Это требует реализации механизмов для парсинга входящих данных, извлечения метрик и их дальнейшей обработки;
- JSON-запросы представляют собой список метрик. Для этого важно было разработать единый формат метрик;
- полученные и обработанные метрики должны сохраняться в базе данных для последующего анализа и отображения. Необходимо обеспечить эффективное взаимодействие с базой данных, включая возможность быстрого выполнения операций вставки, обновления и удаления данных;
- клиентская часть приложения должна предоставлять пользователю интуитивно понятный веб-интерфейс, через который можно

просматривать метрики. Веб-приложение должно включать таблицу, отображающую функции и соответствующие метрики, что позволит пользователю легко воспринимать информацию;

- для повышения удобства работы с метриками необходимо реализовать функции поиска, сортировки и фильтрации данных в таблице;
- необходимо обеспечить высокую производительность приложения, чтобы оно могло обрабатывать большое количество запросов без значительных задержек, особенно при работе с реальным временем;
- приложение должно быть спроектировано с учетом возможного увеличения объема данных и нагрузки.

2.3 Обзор существующих решений

В последние годы наблюдается заметный рост интереса к инструментам мониторинга и управления, что обусловлено стремлением организаций повысить эффективность работы и оптимизировать использование ресурсов. Существующие решения в этой области варьируются от простых систем учета до сложных платформ, интегрирующих различные инструменты и процессы. Одними из наиболее популярных инструментов являются Prometheus и Grafana.

Prometheus[13] — мощная система мониторинга с открытым исходным кодом, разработанная для сбора и хранения метрик в формате временных рядов. Была разработана для мониторинга облачных приложений и микросервисных архитектур.

Grafana[3] — популярный инструмент для визуализации метрик и мониторинга систем. Он интегрируется с различными системами мониторинга и базами данных, в частности с Prometheus, и предоставляет удобный веб-интерфейс для взаимодействия с данными.

Однако архитектура и функциональность данных решений избыточны для нашей задачи, которая требует более узконаправленного и лег-

ковесного подхода с меньшей сложностью в настройке.

2.4 Обзор технологий для реализации приложения

Для реализации приложения в соответствии с поставленными требованиями необходимо определить подходящую систему управления базами данных (СУБД) для эффективного хранения и обработки получаемых от агента данных, а также выбрать технологии для создания клиентской и серверной частей веб-приложения.

2.4.1 База данных

Во время разработки системы для хранения данных возникло ограничение использования реляционных баз данных. Главная причина заключается в том, что изначально неизвестно точное количество метрик, которые могут поступать для обработки и хранения. В реляционных базах данных структура таблиц должна быть четко определена заранее, включая количество столбцов и их типы. Однако, когда количество и тип метрик могут изменяться динамически, это создает сложности для проектирования реляционной базы данных. Необходимость переработки схемы базы данных при каждом изменении структуры привела бы к существенным накладным расходам на поддержку и модификацию системы. В связи с этим было принято решение использовать нереляционную модель "ключ-значение", которая позволяет гибко управлять структурой данных, добавляя новые поля и метрики по мере необходимости без изменения существующей схемы.

В качестве конкретной реализации нереляционной базы данных в нашем проекте мы выбрали JасoDB[5]. Это легковесная и высокопроизводительная база данных, поддерживающая документо-ориентированное хранение данных. Одной из причин этого выбора стало желание протестировать разрабатываемый компонент, который подходил под наши требования, а именно компонент `EntityRelationshipStorage`, который позволяет эффективно управлять сущностями и их взаимосвязями. Этот компонент предоставляет удобный API для работы с дан-

ными. Одной из ключевых особенностей JasoDB является ее способность к простому масштабированию, что делает её идеальным выбором для проектов с непредсказуемым ростом объемов данных. Кроме того, JasoDB обеспечивает высокую скорость обработки запросов, что является критически важным для нашей работы с метриками, которые требуют быстрой агрегации и анализа в режиме, приближенному к реальному времени.

В качестве альтернативы была рассмотрена одна из наиболее популярных документо-ориентированных баз данных — MongoDB[9]. MongoDB предлагает гибкость схемы, позволяя хранить данные в формате JSON, и поддерживает множество функций, включая индексацию и шардирование. Однако, несмотря на свои преимущества, она требует более сложной настройки и администрирования, что может быть трудоемким процессом. Кроме того, при высоких нагрузках могут возникать проблемы с производительностью. В отличие от этого, JasoDB отличается легковесностью и высокой скоростью обработки данных, что делает его более простым в использовании и интеграции, а также удобным для тестирования.

2.4.2 Технологии клиент-серверной части

К выбору фреймворка клиент-серверной части были выдвинуты следующие ключевые требования:

- легковесность,
- высокая производительность,
- простота интеграции с базой данных,
- возможность писать код на одном из хорошо знакомых разработчику языков, таких как Kotlin и Python.

Учитывая данные критерии, были выбраны фреймворки Spring Boot и Ktor.

Spring[12] — фреймворк для создания корпоративных приложений на платформе JVM, предоставляющий широкий спектр инструментов для упрощения разработки.

Ktor[8] — асинхронный веб-фреймворк на языке Kotlin, предназначенный для разработки высокопроизводительных серверных и клиентских приложений с гибкой архитектурой.

Spring предлагает широкий набор встроенных инструментов и готовых решений для множества задач, что делает его идеальным для крупных и сложных проектов. Однако для небольших приложений Spring может быть слишком сложным и требовать больше ресурсов. В то время как Ktor предлагает более простой и легковесный подход, что делает его более подходящим для проектов с меньшей масштабностью и ограниченными ресурсами. В нашем проекте, где важны гибкость, производительность и скорость разработки, Ktor показал себя более подходящим выбором по сравнению с более тяжеловесным Spring.

3 Архитектура

В этом разделе будет описана глобальная архитектура приложения, включающая архитектуру клиентской и серверной частей приложения и их взаимодействие между собой и агентом. Схематично вся архитектура приложения представлена на Рис. 1.

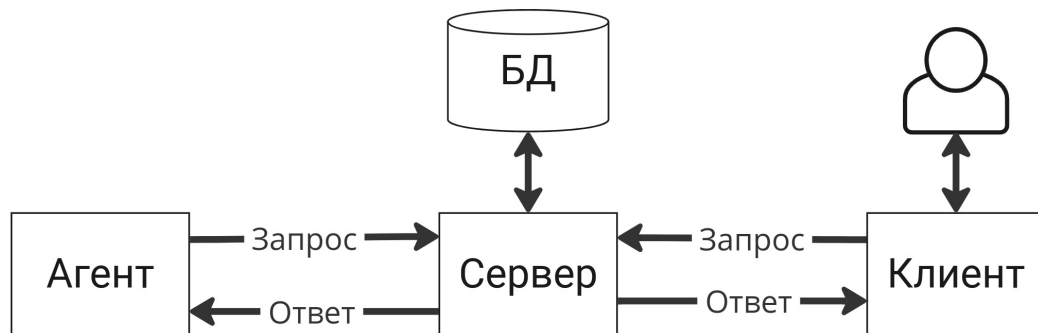


Рис. 1: Архитектура приложения

3.1 Архитектура серверной части

Архитектура серверной части веб-приложения, реализованного с использованием фреймворка Ktor, представляет собой модульную и масштабируемую систему.

Сторонний сервер получает доступ к функциональности серверной части приложения через веб-интерфейс, разработанный в соответствии с архитектурой REST (Representational State Transfer) [2]. В рамках этого подхода клиентская и серверная части разделены, что позволяет им развиваться независимо друг от друга, а взаимодействие между ними происходит посредством стандартных HTTP-методов, таких как GET, PUT, POST и так далее.

MetricStorage является основным компонентом, отвечающим за сохранение и извлечение данных. Оно инкапсулирует логику работы с данными, предоставляя удобный интерфейс для взаимодействия с сервером.

Серверная логика организована в виде маршрутов, которые обрабатывают различные запросы и обеспечивают взаимодействие с хранилищем данных. Набор маршрутов для обработки POST и GET запросов определяется в функции **serverRoutes**. Каждый маршрут приложения отвечает за определенную функциональность, включая сохранение метрик, создание новых сессий и получение уникальных данных.

Приложение использует встроенный сервер на базе Netty[10], который обеспечивает асинхронную обработку запросов и высокую производительность. Это позволяет приложению обрабатывать множество запросов одновременно, что особенно важно для систем мониторинга, где требуется высокая скорость отклика.

Данные передаются в формате JSON, а их сериализация и десериализация осуществляется с помощью плагина **ContentNegotiation**[1], который упрощает работу с JSON-объектами и гарантирует их корректное преобразование.

3.2 Архитектура клиентской части

Клиентская часть представлена в виде одностраничного веб-приложения, которое получает данные, взаимодействуя с серверной стороной приложения.

В основе архитектуры лежит функция **clientRoutes**, которая отвечает за определение маршрутов и обработку запросов. В этой функции инициализируется экземпляр HTTP-клиента, созданного на базе движка CIO, который позволяет отправлять асинхронные запросы к серверу и обрабатывать полученные ответы. HTTP-клиент настроен с помощью плагина **ContentNegotiation**.

После получения данных с сервера клиентская часть генерирует HTML-страницу с таблицей. Для этого используется метод **respondHtml**,

который позволяет динамически формировать HTML-структуру из Kotlin-кода. Для обеспечения актуальности информации на странице используется мета-тег `http-equiv="refresh"`, который обновляет содержимое страницы раз в дельту времени. Это позволяет пользователям видеть изменения в метриках в реальном времени без необходимости вручную перезагружать страницу.

В таблице отображаются имена функций в первом столбце и значения метрик в последующих столбцах. Если для какой-либо метрики нет данных, вместо значения выводится дефис ("-"). Для улучшения визуализации таблицы применяются стили, встроенные в HTML-документ.

3.3 Клиент-серверное взаимодействие

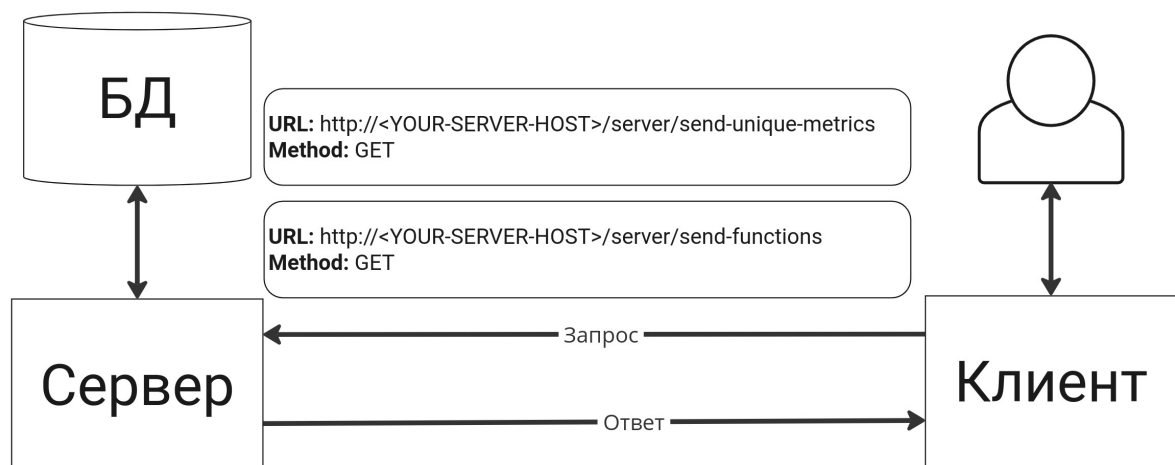


Рис. 2: Клиент-серверное взаимодействие

Взаимодействие между клиентской и серверной частью происходит благодаря `HttpClient`. Когда пользователь переходит по адресу `http://<YOUR-CLIENT-HOST>/client/table`, клиент посылает два асинхронных GET-запроса к серверу:

- Запрос на получение уникальных имен метрик,

- Запрос на список функций, которые необходимо отобразить на странице.

Сервер извлекает нужные данные из базы данных и возвращает их клиенту в формате JSON. Схематично процесс представлен на Рис. 2. После этого клиент десериализует полученные данные и использует их для построения HTML-страницы.

3.4 Взаимодействие с агентом

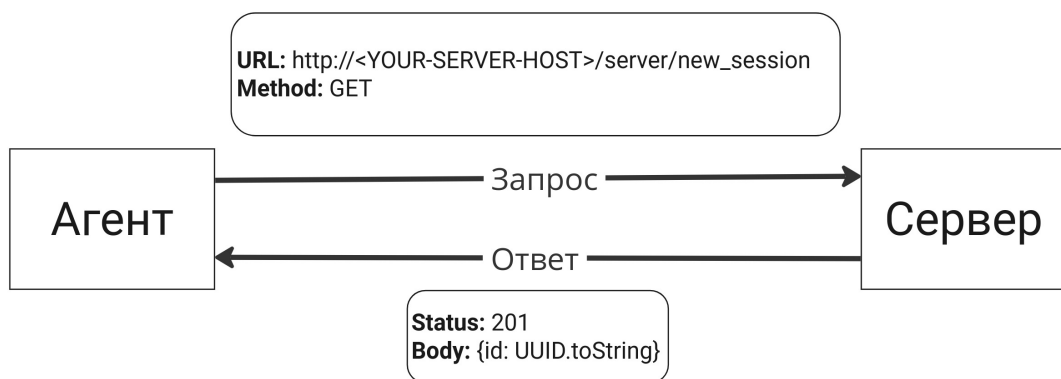


Рис. 3: Процесс получения уникального id для сессии

Агент — это программа, которая собирает метрики с системы и отправляет их на сервер для обработки и хранения.

Взаимодействие между сервером и агентом происходит через два основных маршрута: `/server/save-metric` и `/server/new-session`.

Когда агент запускается или начинает новую сессию, он отправляет GET запрос по маршруту `/server/new-session`. Сервер генерирует уникальный идентификатор сессии в формате UUID, который затем отправляется агенту в ответ с кодом статуса 200 OK. Агент сохраняет полученный UUID и использует его для маркировки всех дальнейших метрик, которые он отправляет во время этой сессии. Этот уникальный идентификатор позволяет различать данные, поступающие от одного или нескольких агентов, и связывать их с определенными сессиями мониторинга. Схематично этот процесс представлен на Рис. 3.

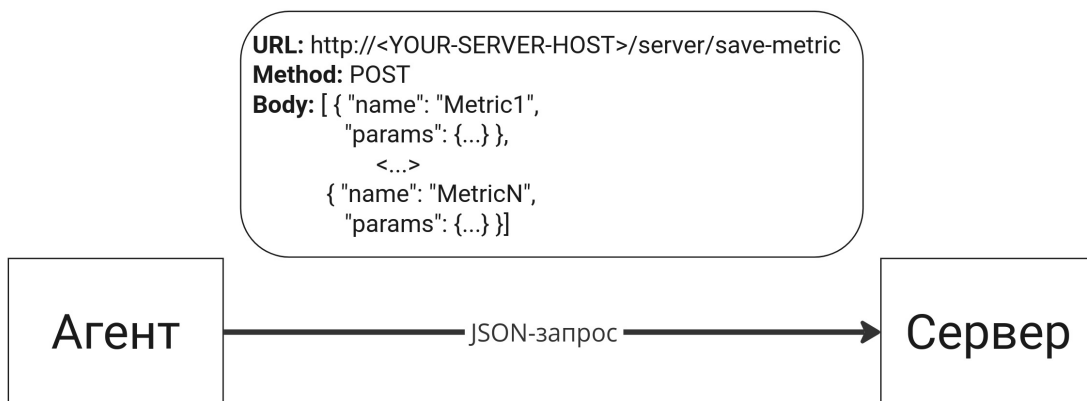


Рис. 4: Процесс отправки метрик

После того как агент получил уникальный идентификатор сессии, он начинает собирать метрики с системы. Эти данные агент отправляет на сервер по маршруту `/server/save-metric` (Рис. 4). В теле запроса содержится JSON-массив, который представляет собой список метрик. Сервер получает этот запрос, десериализует JSON в объекты типа `Metric` с помощью встроенной системы сериализации. Полученные объекты сохраняются в базу данных, а в ответ агент получает сообщение о том, что метрики успешно обработаны и сохранены.

4 Детали реализации

4.1 Формат метрик

JSON-запросы от агента представляют собой список метрик, что требует их структурированного и последовательного представления. Для обеспечения удобства обработки и анализа данных, а также для унификации процесса сбора и передачи информации, было решено разработать единый формат метрик.

Формат данных, передаваемых на сервер, представлен в виде JSON-структуры, которая включает массив объектов. Каждый объект описывает отдельную метрику и состоит из двух основных компонентов: поля **name**, обозначающего имя метрики, и вложенного объекта **params**, содержащего набор параметров метрики.

Параметры метрики включают следующие поля:

- **funName** — задает имя функции;
- **type** — указывает тип данных метрики;
- **value** — представляет значение, соответствующее указанному типу;
- **transitive** — булевый параметр, который указывает на транзитивность метрики.

Тип данных метрики (**type**) может быть представлен одним из следующих значений: `Int`, `Double`, `String` или `Boolean`, причём значение поля **value** должно строго соответствовать указанному типу.

Массив может содержать произвольное количество метрик, каждая из которых описывается аналогичной структурой, что позволяет передавать на сервер несколько метрик в одном запросе.

4.2 JacoDB и RAMEntityRelationshipStorage

Веб-приложение использует JacoDB в качестве базы данных, а конкретно — реализацию хранилища `RAMEntityRelationshipStorage`, что-

бы управлять метриками и функциями. Хранилище создается с помощью метода `newStorage`, который инициализирует базу данных для каждой сессии, как экземпляр `RAMEntityRelationshipStorageSPI`, который хранит данные в оперативной памяти, обеспечивая высокую скорость доступа и обработки. Поскольку хранилище RAM, оно не сохраняет данные на диск, что делает его подходящим для временных или тестовых данных. В дальнейшем можно будет использовать другие хранилища реализующие интерфейс `EntityRelationshipStorage`. Например, `SqlEntityRelationshipStorage`, который позволяет сохранять данные в реляционной базе данных. Подобную гибкость обеспечивает единый интерфейс взаимодействия.

Все операции с хранилищем выполняются внутри транзакции, что гарантирует целостность данных. Метод `transactional` обеспечивает атомарность операций, позволяя откатывать изменения в случае возникновения ошибок.

В базе данных реализовано несколько функций, ключевыми являются:

- сохранение метрик — создает новую сущность `Function` или обновляет существующую, добавляя новую метрику;
- получение уникальных имен метрик — функция, которая позволяет извлекать свойства всех сущностей типа `Function`. Полученные данные используются в клиентской части для отрисовки столбцов таблицы;
- получение функций по страницам — извлекает все сущности типа `Function` для заданной страницы, основываясь на номере страницы и количестве элементов, которое задается константой `NUM OF ELEMENT ON PAGE`.

5 Тестирование

В рамках разработки системы мониторинга был реализован процесс тестирования, направленный на проверку его устойчивости и корректности обработки данных. Для этого был разработан mock объект, который имитирует работу агента.

Процесс генерации метрик осуществляется с помощью функции `generateRandomMetric()`, которая создает объект `Metric`. Генерация значений осуществляется случайным образом, что позволяет имитировать разнообразие данных, которые система может обрабатывать.

Для отправки сгенерированных метрик на сервер была разработана функция `sendMetrics`, которая устанавливает HTTP-соединение и отправляет данные в формате JSON. Код ответа сервера выводится в консоль, чтобы отслеживать успешность выполнения запросов. В ходе тестирования через фиксированный интервал времени происходила отправка случайного количества метрик (от 1 до 30).

Данные, полученные в результате работы, были проверены путем визуального анализа с помощью веб-интерфейса. Веб-интерфейс корректно отображал функции и их метрики, позволяя оценить ожидаемую работу клиент-серверного приложения.

Заключение

В рамках данной работы были получены следующие результаты

- описаны технологии для создания веб-приложения, реализующих функциональность мониторинг системы;
- сформулированы требования;
- разработана архитектура на основе выявленных требований;
- реализована мониторинг система;
- проведено тестирование на генерируемых тестовых данных.

Ссылка на репозиторий с реализацией:

<https://github.com/juliakononov/monitoring>

Список литературы

- [1] Content negotiation and serialization. — URL: <https://ktor.io/docs/client-serialization.html>.
- [2] Fielding Roy Thomas. Architectural styles and the design of network-based software architectures. — University of California, Irvine, 2000.
- [3] Grafana documentation. — URL: <https://grafana.com/docs/grafana/latest/>.
- [4] Ivanov Dmitry. KLEEF: Symbolic Execution Engine (Competition Contribution) // Fundamental Approaches to Software Engineering: 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings / Springer Nature. — Vol. 14573. — 2024. — P. 314.
- [5] JacoDB. — URL: <https://github.com/UnitTestBot/jacodb>.
- [6] KLEE. — 2024. — URL: <https://klee-se.org/docs/>.
- [7] KLEEF. — URL: <https://github.com/UnitTestBot/klee>.
- [8] Ktor Documentation. — 2024. — URL: <https://ktor.io/docs/welcome.html>.
- [9] MongoDB. — URL: <https://www.mongodb.com/>.
- [10] Netty. — 2024. — URL: <https://netty.io/wiki/index.html>.
- [11] Niemetz Aina, Preiner Mathias. Bitwuzla // International Conference on Computer Aided Verification / Springer. — 2023. — P. 3–17.
- [12] Spring Boot. — URL: <https://spring.io/projects/spring-boot>.
- [13] What is Prometheus? — URL: <https://prometheus.io/docs/introduction/overview/>.

- [14] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.