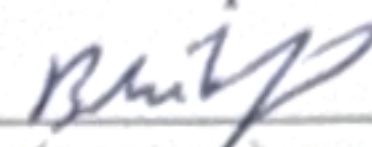


БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информационных технологий и робототехники
Кафедра «Программное обеспечение информационных систем и технологий»

ДОПУЩЕН К ЗАЩИТЕ

Старший преподаватель


(подпись)

Н.В. Воюш
(инициалы и фамилия)

« 02 » 06 2026 г.

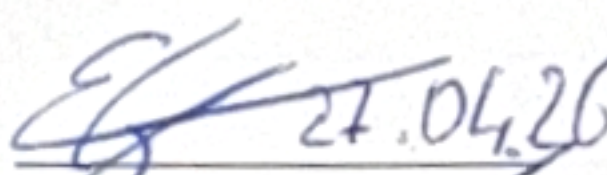
РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
ДИПЛОМНОГО ПРОЕКТА

«Программное средство нагрузочного тестирования WEB-приложений в
изолированной контейнерной сред»

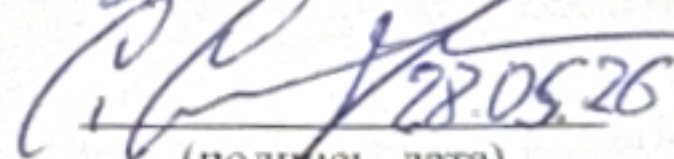
Специальность 1-40 01 01 «Программное обеспечение информационных
технологий»

Специализация 1-40 01 01 05 «Управление качеством и тестирование
программного обеспечения»

Обучающийся
группы 10701122
(номер)

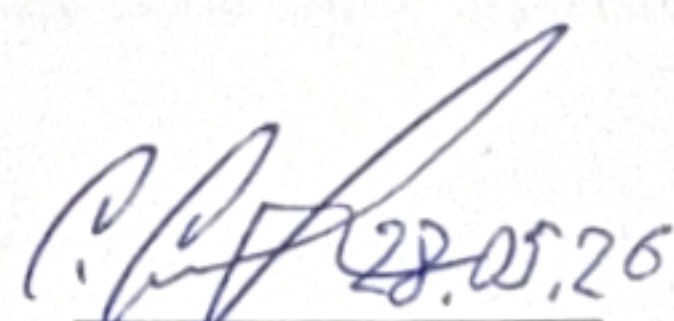

(подпись, дата) Е.Д. Ханяк

Руководитель

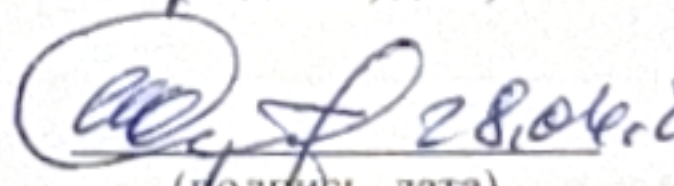

(подпись, дата) С.Ю. Станкевич

Консультанты:

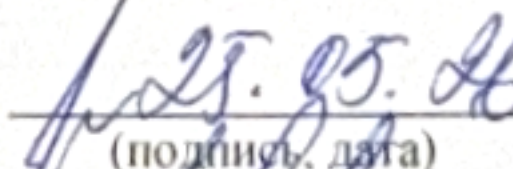
по разделу «Компьютерное
проектирование»


(подпись, дата) С.Ю. Станкевич

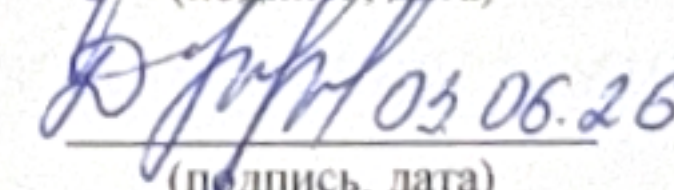
по разделу «Охрана труда»


(подпись, дата) А.М. Лазаренков

по разделу «Экономика»


(подпись, дата) Т.Н. Беляцкая

Ответственный за нормоконтроль


(подпись, дата) Н.С. Домаренко

Объем проекта:

расчетно-пояснительная записка – 105 страниц;

графическая часть – 10 листов;

магнитные (цифровые) носители – 1 единиц.

Минск 2026

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники

Кафедра «Программное обеспечение информационных систем и технологий»

УТВЕРЖДАЮ

Старший преподаватель

(подпись) Н.В. Воюш
(инициалы и фамилия)

« ____ » _____ 2026 г.

Задание на дипломный проект

Обучающемуся Ханяку Е.Д

(фамилия, инициалы)

1. Тема дипломного проекта: Программное средство нагрузочного тестирования
WEB-приложений в изолированной контейнерной среде

Утверждена приказом руководителя высшего образования от 17.04.2026 № 1230-лс

2. Исходные данные к дипломному проекту:

Среда развертывания – Linux-серверы и контейнеры.

Языки программирования – Go 1.26, JavaScript.

Среда разработки – Golang.

Фреймворки и библиотеки – chi, sqlc, goose, React, Vite, Recharts.

СУБД – SQLite

Средства контейнеризации и развертывания – Docker, Docker compose.

3. Перечень подлежащих разработке вопросов или краткое содержание расчетно-пояснительной записки:

– введение;

– анализ предметной области;

– постановка задачи;

– методические основы разработки приложения;

– программная реализация системы;

– руководство пользователя;

– технико-экономическое обоснование разработки программного обеспечения;

– охрана труда;

– заключение;

– список использованных источников;

– приложение.

4. Перечень графического материала (с точным указанием обязательных чертежей и графиков)

– результат анализа предметной области (1-2 слайда);

– результат проектирования (5-7 слайдов);

– результат реализации программного обеспечения (2-3 слайда);

– результат тестирования программного обеспечения (1-2 слайда);

– результат прогонки контрольного примера (2-3 слайда);

– технико-экономические показатели (1 слайд).

5. Консультанты по дипломному проекту с указанием относящихся к ним разделов:

по компьютерному проектированию – С.Н. Станкевич

по охране труда – А.М. Лазаренков

по вопросам экономики – Т.Н. Беляцкая

нормоконтроль – Н.С. Домаренко

6. Примерный календарный график выполнения дипломного проекта:

Наименование этапов выполнения дипломного проекта, содержание расчетно-пояснительной записки, графического материала	Объем работы, %	Сроки (дата) выполнения этапа	Примечание (в т.ч. отметка руководителя, консультанта о выполнении)
Обзор и изучение литературы по теме дипломного проекта, проектирование приложения	50%	24.04.2026	
Реализация приложения	70%	11.05.2026	
Тестирование, разработка вопросов экономики и охраны труда	90%	22.05.2026	
Подготовка пояснительной записки и графической части дипломного проекта	100%	29.05.2026	

7. Дата выдачи задания

09.02.2026

8. Срок сдачи законченного дипломного проекта

29.05.2026

Руководитель

(подпись)

С.Н. Станкевич

(инициалы, фамилия)

Подпись

обучающегося

(подпись)

Е.Д. Ханяк

(инициалы, фамилия)

Дата

09.02.2026

РЕФЕРАТ

НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ, РАСПРЕДЕЛЁННЫЙ АГЕНТ, УПРАВЛЯЮЩАЯ ПЛАТФОРМА, ВИРТУАЛЬНЫЙ ПОЛЬЗОВАТЕЛЬ, HTTP-СЦЕНАРИЙ, МЕТРИКИ ПРОИЗВОДИТЕЛЬНОСТИ

Объектом разработки является распределённая система нагрузочного тестирования веб-приложений LoadForge, построенная по архитектуре «control plane – data plane».

Цель проекта – разработка автономного агента генерации нагрузки и управляющей платформы с веб-интерфейсом, обеспечивающих выполнение параметризованных HTTP-сценариев с условными переходами и сбор метрик производительности.

В процессе проектирования проведён анализ инструментов Apache JMeter, Locust и k6; спроектирована архитектура «control plane – data plane»; реализованы агент на Go (cobra, yaml.v3, gjson, kin-openapi, net/http) и управляющая платформа на Go (chi, sqlc, SQLite, JWT) с веб-клиентом на React; проведено модульное, функциональное, интеграционное и нагрузочное тестирование.

Элементами практической значимости являются лёгкий агент с декларативным описанием сценариев на YAML, нативная интеграция с платформой по REST API, горизонтальная масштабируемость генерации нагрузки и подтверждённая производительность платформы (свыше 10 000 запросов в секунду при практически нулевой доле ошибок).

Областью возможного практического применения являются организации, разрабатывающие и эксплуатирующие высоконагруженные веб-сервисы и API, в том числе в составе процессов CI/CD.

В ходе дипломного проектирования прошли апробацию архитектура «control plane – data plane», двухэтапная аутентификация агента «enrollment-токен → session-токен», механизм длинного опроса с публикацией заданий через RunBus и нагрузочное тестирование платформы инструментом k6.

Результатами внедрения явились автоматизация распределённого нагрузочного тестирования веб-сервисов, снижение трудозатрат и повышение достоверности получаемых результатов.

Студент-дипломник подтверждает, что приведённый в дипломном проекте расчётно-аналитический материал объективно отражает состояние исследуемого процесса (разрабатываемого объекта), все заимствованные из литературных и других источников теоретические и методологические положения и концепции сопровождаются ссылками на их авторов.

Дипломный проект: 105 с., 19 рис., 25 табл., 39 источников, 1 прил.

№ строки	Формат	Обозначение			Наименование	Кол. листов	Примечание	
1					Документация общая			
2								
3	A4	–			Задание по дипломному проекту	1		
4	A4	ДП–10702122/12–2026–РПЗ			Расчётно–пояснительная записка	105		
5	A4	ДП–10706122/22–2026–01			Схема взаимодействия компонентов проектируемой системы	1		
6	A4	ДП–10706122/22–2026–02			Диаграмма вариантов использования	1		
7	A4	ДП–10706122/22–2026–03			Результаты нагрузочного тестирования	1		
8	A4	ДП–10706122/22–2026–04			Структура проекта	1		
9	A4	ДП–10706122/22–2026–05			Окно регистрации в системе	1		
10	A4	ДП–10706122/22–2026–06			Графический редактор сценариев	1		
11	A4	ДП–10706122/22–2026–07			Раздел «Настройки»	1		
12	A4	ДП–10706122/22–2026–08			Расчет затрат на основную заработную плату команды разработчиков	1		
13	A4	ДП–10706122/22–2026–09			Сводные технико-экономические показатели проекта	1		
14	A4	ДП–10706122/22–2026–10			Предельно-допустимые уровни звука, эквивалентные уровни звука и уровни звукового давления в октавных полосах частот при работе с ВДТ, ЭВМ и ПЭВМ и периферийными устройствами	1		
Изм.	Лист	№ документа	Подпись	Дата	ДП–10706122/22–2026–РПЗ			
Разраб.		Ханяк			Ведомость объема дипломного проекта	Лит.	Лист	Листов
Руковод.		Станкевич				У	1	1
Консульт.		Станкевич				1–40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						
Утверд.		Воюш						

ОГЛАВЛЕНИЕ

ВЕДОМОСТЬ ОБЪЕМА ДИПЛОМНОГО ПРОЕКТА	
ВВЕДЕНИЕ	8
1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ.....	9
1.1 Эволюция систем нагрузочного тестирования	9
1.2 Обзор существующих решений.....	11
1.2.1 Инструмент нагрузочного тестирования Apache JMeter	12
1.2.2 Фреймворк нагрузочного тестирования Locust	13
1.2.3 Инструмент нагрузочного тестирования k6.....	15
1.3 Недостатки существующих решений	16
2 ПОСТАНОВКА ЗАДАЧИ.....	18
2.1 Архитектура проектируемой системы	19
2.2 Варианты использования.....	21
3 МЕТОДИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ПРИЛОЖЕНИЯ	24
3.1 Выбор и обоснование архитектуры агента.....	24
3.2 Выбор средств разработки для реализации системы	25
3.2.1 Язык программирования Go	25
3.2.2 Библиотека командной строки Cobra.....	27
3.2.3 Стандартная библиотека net/http	27
3.2.4 Библиотека gopkg.in/yaml.v3 для разбора сценариев.....	28
3.2.5 Библиотека gjson для извлечения данных из ответов	29
3.2.6 Библиотека kin-openapi для валидации сценариев	29
3.2.7 Docker и контейнеризация	30
4 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ	31
4.1 Структура проекта.....	31
4.2 Обоснование выбранной схемы контейнеризации компонентов	33
4.3 Описание компонентов агента	34
4.3.1 Командно-строчный интерфейс агента.....	35
4.3.2 Подсистема описания тестовых сценариев.....	35
4.3.3 Низкоуровневый исполнитель HTTP-запросов	36
4.3.4 Ядро генерации нагрузки	37
4.3.5 Клиент управляющей платформы	38
4.4 Описание сервисов управляющей платформы	38
4.4.1 Авторизационный сервис (AuthService)	39
4.4.2 Управление проектами (ProjectService)	39
4.4.3 Аутентификация агентов на платформе (AgentService).....	39
4.4.4 Управление тестовыми сценариями (ScenarioService).....	40
4.4.5 Реализация тестового сценария (TestRunService).....	40
4.4.6 Метрики (MetricService).....	41

4.5 Интерфейсы и взаимодействия.....	41
4.5.1 Эндпоинты аутентификации.....	42
4.5.2 Эндпоинты управления проектами.....	43
4.5.3 Эндпоинты управления агентами.....	44
4.5.4 Эндпоинты управления тестовыми сценариями.....	44
4.5.5 Эндпоинты управления тестовыми запусками.....	45
4.5.6 Агентские эндпоинты.....	46
4.6 Тестирование разработанной системы.....	48
4.6.1 Виды тестирования и применяемые подходы.....	48
4.6.2 Модульное тестирование агента.....	49
4.6.3 Функциональное тестирование управляющей платформы.....	52
4.6.4 Интеграционное тестирование.....	55
4.6.5 Нагрузочное тестирование платформы средствами k6.....	56
4.6.6 Результаты тестирования.....	58
5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ.....	59
5.1 Назначение и состав системы.....	59
5.2 Системные требования и установка.....	59
5.3 Регистрация и вход в систему.....	60
5.4 Структура главного окна.....	62
5.5 Раздел «Главная».....	62
5.6 Раздел «Проекты».....	64
5.7 Раздел «Агенты».....	66
5.8 Подключение агента к платформе.....	68
5.9 Раздел «Сценарии».....	68
5.9.1 Импорт OpenAPI-спецификации.....	70
5.9.2 Графический редактор шагов.....	71
5.9.3 Свойства узла.....	71
5.9.4 Условные переходы между шагами.....	72
5.9.5 Просмотр YAML-эквивалента и общие настройки сценария.....	72
5.9.6 Сохранение и загрузка сценария.....	74
5.10 Раздел «Запуски».....	75
5.11 Просмотр и анализ результатов запуска.....	76
5.12 Раздел «Настройки».....	78
5.13 Сообщения об ошибках.....	80
6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....	82
6.1 Характеристика разработанного по индивидуальному заказу программного средства.....	82
6.2 Расчет затрат на разработку и цена программного средства, созданного по индивидуальному заказу.....	83

6.3 Оценка экономического эффекта для организации-заказчика	88
6.4 Расчет показателей эффективности инвестиций в разработку программного средства.....	90
7 ОХРАНА ТРУДА	95
7.1 Производственная санитария, техника безопасности и пожарная профилактика.....	95
7.1.1 Метеоусловия	95
7.1.2 Вентиляция и отопление	96
7.1.3 Освещение	97
7.1.4 Шум	98
7.1.5 Электробезопасность.....	99
7.1.6 Излучение	100
7.1.7 Пожарная безопасность.....	101
7.2 Требования к помещениям для работы с ВДТ, ЭВМ и ПЭВМ.....	102
ЗАКЛЮЧЕНИЕ	104
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	
ГРАФИЧЕСКАЯ ЧАСТЬ	
ПРИЛОЖЕНИЕ А	

ВВЕДЕНИЕ

В наше время обеспечение стабильности и высокой производительности программных систем становится одной из ключевых задач в сфере разработки программного обеспечения. С ростом числа пользователей и увеличением нагрузки на серверные системы возникает острая необходимость в инструментах, позволяющих заблаговременно выявлять узкие места и оценивать пределы производительности приложений. Нагрузочное тестирование является неотъемлемой частью обеспечения качества программных продуктов, позволяя моделировать реальные условия эксплуатации и принимать обоснованные архитектурные решения.

Традиционно нагрузочное тестирование требует значительных ресурсов и ручной настройки тестовых сценариев, что приводит к высоким трудозатратам, сложности масштабирования и неоднородности получаемых результатов. В современных условиях актуальным становится создание распределённых агентов нагрузочного тестирования, способных выполнять тестовые сценарии параллельно и автономно, собирая метрики в режиме реального времени и передавая их в управляющую систему.

Разработка подобного программного обеспечения предполагает решение следующих задач:

- 1) анализ существующих подходов к нагрузочному тестированию и распределённым системам;
- 2) проектирование архитектурного агента, обеспечивающей высокую производительность и масштабируемость;
- 3) разработка механизма выполнения HTTP-сценариев с поддержкой переменных, условных переходов и извлечения данных от ответов;
- 4) реализация модуля сбора и передачи метрик в управляющую платформу;
- 5) тестирование разработанного агента в условиях распределённой нагрузки;
- 6) разработка документации и методических рекомендаций по использованию системы.

Решение перечисленных задач позволит создать легкий и высокопроизводительный агент нагрузочного тестирования, снижающий трудозатраты на подготовку и проведение нагрузочных испытаний и повышающий достоверность получаемых результатов.

Целью проекта является разработка распределённого агента нагрузочного тестирования, выполняющего тестовые сценарии и передающего метрики производительности в управляющую платформу.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Эволюция систем нагрузочного тестирования

Эволюция систем нагрузочного тестирования тесно связана с развитием веб-технологий, ростом объёмов интернет-трафика и повышением требований к надёжности программных систем. С момента появления первых инструментов автоматизированной проверки производительности данная область прошла путь от простейших утилит командной строки до распределённых облачных платформ, способных генерировать нагрузку, эквивалентную миллионам реальных пользователей.

На начальном этапе – в конце 1990-х годов – тестирование производительности осуществлялось преимущественно вручную: специалисты анализировали логи сервера, измеряли время отклика с помощью сетевых снифферов и создавали простые скрипты для имитации запросов. Первые специализированные инструменты, такие как Apache JMeter (1998) и Microsoft Web Application Stress Tool, появились именно в этот период. Их основная задача сводилась к генерации HTTP-запросов по заданному сценарию и измерению времени ответа, однако они не поддерживали распределённое выполнение и требовали значительных вычислительных ресурсов на одной машине.

Настоящим импульсом к развитию данной области стало массовое распространение электронной коммерции, социальных сетей и SaaS-сервисов в 2000-х годах. Рост числа одновременных пользователей сделал критически важным понимание того, как система ведёт себя под нагрузкой ещё до её вывода в промышленную эксплуатацию. Это потребовало инструментов, способных моделировать тысячи виртуальных пользователей одновременно, воспроизводить сложные пользовательские сценарии с авторизацией, навигацией и передачей данных, а также собирать детальную метрику в реальном времени.

С середины 2000-х годов начался активный переход к сценарно-ориентированному тестированию. Инструменты стали поддерживать запись пользовательских сессий, параметризацию запросов, работу с переменными, условные переходы между шагами и корреляцию данных между запросами (извлечение токенов, идентификаторов сессий и т.д.). Именно в этот период сформировалось современное представление о нагрузочном тесте как о воспроизводимом, параметризованном сценарии, охватывающем полный жизненный цикл взаимодействия пользователя с системой.

Следующий этап – переход к распределённому тестированию. С ростом масштабов систем одна машина перестала справляться с генерацией

достаточного числа виртуальных пользователей. Появились архитектуры «master–agent», в которых управляющий узел координирует работу множества агентов, равномерно распределяя нагрузку. Такой подход позволил проводить тесты с десятками тысяч одновременных пользователей без необходимости использования дорогостоящего монолитного оборудования.

С приходом эпохи DevOps и методологий Continuous Integration / Continuous Delivery нагрузочное тестирование стало неотъемлемой частью конвейера разработки программного обеспечения. Инструменты нового поколения – Locust, k6, Gatling – разрабатывались с прицелом на интеграцию в CI/CD-пайплайны: лёгкость запуска из командной строки, поддержка кода как конфигурации (code-as-config), возможность задавать пороговые значения метрик и автоматически прерывать тест при их превышении.

В 2010-х годах широкое распространение получили облачные платформы нагрузочного тестирования – BlazeMeter, Flood.io. Они предоставили командам разработчиков возможность запускать масштабные тесты без необходимости поддерживать собственную инфраструктуру агентов: облачный провайдер брал на себя задачи по выделению вычислительных ресурсов, распределению нагрузки и агрегации метрик. Результаты тестов стали доступны через удобные веб-интерфейсы с интерактивными графиками и историей запусков.

Современные системы нагрузочного тестирования стали значительно интеллектуальнее. Они поддерживают адаптивные сценарии с динамической генерацией данных, интеграцию с системами мониторинга (Prometheus, Grafana, Datadog), автоматический анализ регрессий производительности между запусками и сравнение профилей нагрузки. Появилась концепция хаос-инжиниринга, расширяющая понятие нагрузочного теста: помимо проверки производительности, системы намеренно вводят ошибки и задержки для оценки устойчивости к сбоям.

Одним из важнейших изменений последних лет стал сдвиг от монолитных инструментов к лёгким, специализированным агентам. Вместо единого тяжёлого приложения, выполняющего все функции сразу, современные архитектуры предполагают разделение ответственности: управляющая платформа (control plane) задаёт сценарий, регистрирует агентов и собирает метрики, а лёгкие агенты (data plane) выполняют HTTP-запросы и передают результаты. Такой подход существенно упрощает горизонтальное масштабирование, позволяет запускать агенты в Docker- и Kubernetes-окружениях и снижает порог входа для интеграции с облачными провайдерами.

Таким образом, современная система нагрузочного тестирования – это не просто генератор запросов, а комплексный инструмент, который:

- 1) выполняет параметризованные сценарии с поддержкой переменных, условных переходов и извлечения данных из ответов;
- 2) работает в распределённом режиме, координируя множество агентов;
- 3) собирает метрики в реальном времени и передаёт их в управляющую систему;
- 4) интегрируется с инфраструктурой CI/CD и облачными платформами;
- 5) поддерживает различные протоколы – HTTP/HTTPS, WebSocket, gRPC и другие.

Наряду с перечисленными функциональными возможностями значимым изменением последнего десятилетия стало смещение акцента с разовых нагрузочных испытаний на непрерывное тестирование производительности. В рамках практик Site Reliability Engineering нагрузочные тесты выполняются регулярно как часть процесса релиза, а получаемые метрики (RPS, время отклика по перцентилям, частота ошибок) сравниваются с целевыми показателями уровня обслуживания – SLO и SLI. Такой подход требует, чтобы инструмент нагрузочного тестирования был лёгким, быстро запускающимся и пригодным для встраивания в автоматизированные конвейеры без необходимости развёртывания тяжёлой инфраструктуры.

Не менее важной тенденцией является сближение нагрузочного тестирования с подсистемами наблюдаемости (observability). Современные агенты передают не только агрегированные показатели, но и подробные трассировки (distributed traces) и структурированные журналы, что позволяет соотносить деградацию производительности с конкретными участками кода тестируемого сервиса. Применение единого формата метрик OpenTelemetry, поддержка экспортёров Prometheus и совместимость с инструментами визуализации Grafana становятся фактически отраслевым стандартом, без поддержки которых современный инструмент не может считаться конкурентоспособным.

Однако, несмотря на очевидный прогресс, существующие решения нередко страдают избыточной сложностью, высоким потреблением ресурсов и недостаточной гибкостью сценарного описания. Именно на этом фоне возникает потребность в разработке лёгкого, высокопроизводительного агента, способного эффективно выполнять сложные тестовые сценарии и бесшовно интегрироваться с управляющей платформой.

1.2 Обзор существующих решений

Современный рынок инструментов нагрузочного тестирования охватывает широкий спектр решений – от классических инструментов с графическим интерфейсом до современных фреймворков, ориентированных

на программирование сценариев и интеграцию в DevOps-процессы. Каждый из них представляет собой отдельный подход к организации нагрузочного тестирования и отличается по архитектуре, производительности, удобству настройки и применимости в различных контекстах.

Анализ существующих решений важен как для понимания общих тенденций развития отрасли, так и для обоснования архитектурных решений, принятых при разработке собственного агента. Ниже рассмотрены три наиболее распространённых инструмента, представляющих различные подходы к нагрузочному тестированию: Apache JMeter, Locust и k6. Каждый из них занимает определённую нишу и обладает выраженными преимуществами и ограничениями.

1.2.1 Инструмент нагрузочного тестирования Apache JMeter

Apache JMeter – один из старейших и наиболее широко применяемых инструментов нагрузочного тестирования с открытым исходным кодом. Разработанный компанией Apache Software Foundation и впервые выпущенный в 1998 году, JMeter изначально создавался для тестирования веб-приложений, однако со временем превратился в универсальный инструмент, поддерживающий широкий спектр протоколов: HTTP, HTTPS, FTP, JDBC, LDAP, SOAP, JMS и ряд других.

Архитектурно JMeter представляет собой Java-приложение с графическим интерфейсом пользователя на базе Swing, что обеспечивает кроссплатформенность и широкую совместимость, однако налагает требование к наличию установленной Java-среды выполнения на каждой машине, задействованной в тестировании. Графический интерфейс позволяет визуально конструировать тестовые планы, добавляя элементы – группы потоков, сэмплы, контроллеры, слушатели – в виде дерева компонентов.

Одной из ключевых особенностей JMeter является визуальный редактор тестовых планов. Специалист по тестированию может создать полный сценарий без написания кода, используя встроенные элементы: HTTP-запросы, препроцессоры, постпроцессоры, регулярные выражения для извлечения данных, assertion-проверки корректности ответов и таймеры задержки. Это снижает порог входа и делает инструмент доступным для специалистов без опыта программирования.

Однако JMeter обладает рядом существенных ограничений. Основной из них – высокое потребление ресурсов. Каждый виртуальный пользователь в JMeter реализован как отдельный Java-поток (thread), что накладывает жёсткое ограничение на количество одновременных пользователей на одной машине. При тестировании с тысячами виртуальных пользователей JMeter требует

значительного объёма оперативной памяти и процессорных ресурсов, что существенно увеличивает стоимость инфраструктуры.

Графический интерфейс, являющийся одновременно преимуществом и недостатком, создаёт неудобства при автоматизации и интеграции в CI/CD-пайплайны. Тестовые планы хранятся в формате XML (.jmx), который плохо поддаётся ревью в системах контроля версий и ручному редактированию.

Также следует отметить, что JMeter изначально проектировался как синхронный, потоково-ориентированный инструмент, что делает его менее эффективным по сравнению с современными асинхронными решениями при одинаковом числе виртуальных пользователей. Это особенно заметно при тестировании современных высоконагруженных API, где критически важна способность агента обрабатывать большое число одновременных соединений при минимальных собственных затратах ресурсов.

Таким образом, Apache JMeter остаётся мощным и функционально богатым инструментом, особенно ценным в корпоративных средах, где требуется широкая поддержка протоколов и визуальное создание сценариев. Однако его архитектура, ориентированная на модель «один поток – один пользователь», ограничивает масштабируемость и делает его менее подходящим для современных высоконагруженных сценариев тестирования, требующих эффективного использования ресурсов и глубокой интеграции с DevOps-процессами.

1.2.2 Фреймворк нагрузочного тестирования Locust

Locust – это современный фреймворк нагрузочного тестирования с открытым исходным кодом, написанный на языке Python. Проект был создан в 2011 году и быстро завоевал популярность среди разработчиков благодаря принципиально иному подходу к описанию тестовых сценариев: вместо XML-конфигураций или визуального редактора Locust использует обычный Python-код, что делает сценарии максимально гибкими и легко поддерживаемыми в системах контроля версий.

Ключевая идея Locust состоит в том, что виртуальный пользователь описывается как Python-класс, наследующий от `HttpUser`. Методы этого класса, помеченные декоратором `@task`, определяют действия, которые пользователь выполняет в ходе теста. Такой подход позволяет использовать всю мощь языка Python: условные ветвления, циклы, работу с библиотеками, генерацию случайных данных, извлечение значений из ответов и их передачу в последующие запросы.

Архитектурно Locust реализует виртуальных пользователей как лёгкие зелёные потоки (greenlets) на базе библиотеки `gevent`, что позволяет запускать

тысячи одновременных пользователей при значительно меньшем потреблении памяти по сравнению с потоково-ориентированными решениями, такими как JMeter. Асинхронная природа greenlets обеспечивает высокую эффективность при тестировании I/O-интенсивных API.

Locust поставляется со встроенным веб-интерфейсом, доступным в браузере в режиме реального времени. Через него можно запустить тест, указать количество пользователей и скорость их нарастания (ramp-up rate), наблюдать за текущими значениями RPS, временем ответа и процентом ошибок, а также останавливать и перезапускать тест без перезапуска самого приложения.

Распределённый режим в Locust реализован по схеме master–worker. Главный узел управляет веб-интерфейсом и агрегирует статистику, а worker-процессы непосредственно генерируют нагрузку. Распределение выполняется через встроенный механизм коммуникации на базе ZeroMQ, что обеспечивает эффективную передачу команд и результатов между узлами. Запуск в распределённом режиме не требует сложной настройки и легко реализуется как локально, так и в Kubernetes-кластере.

Поддержка расширений и пользовательских слушателей событий делает Locust гибкой платформой для интеграции с внешними системами. Разработчики могут подписываться на события запросов, ответов и ошибок для экспорта метрик в Prometheus, InfluxDB, Elasticsearch и другие системы мониторинга. Также доступны готовые расширения сообщества, добавляющие поддержку дополнительных протоколов, геймифицированных отчётов и интеграции с CI/CD.

Вместе с тем Locust обладает рядом ограничений. Поскольку сценарии пишутся на Python, использование инструмента требует уверенного владения языком программирования, что повышает порог входа для специалистов по тестированию без опыта разработки. Кроме того, производительность Locust ограничена возможностями интерпретатора CPython и библиотеки `gevent`: при экстремально высокой нагрузке (сотни тысяч RPS) инструмент может уступать компилируемым аналогам.

Встроенная система отчётности Locust достаточно лаконична: веб-интерфейс отображает агрегированную статистику по конечным точкам, но не предоставляет детальных гистограмм распределения задержек, построения произвольных дашбордов или экспорта в формат HTML без дополнительных плагинов. Для получения полноценной аналитики необходима интеграция с внешними системами мониторинга.

Таким образом, Locust является отличным выбором для команд разработчиков, знакомых с Python и стремящихся к максимальной гибкости сценариев при умеренных требованиях к масштабу нагрузки. Однако

необходимость программирования на Python и относительно скромные возможности встроенной отчётности делают его менее подходящим для организаций, предпочитающих декларативный подход к описанию тестовых сценариев.

1.2.3 Инструмент нагрузочного тестирования k6

k6 – современный инструмент нагрузочного тестирования с открытым исходным кодом, разработанный компанией Grafana Labs. Проект был создан в 2017 году и изначально проектировался с прицелом на интеграцию в DevOps-процессы, использование в CI/CD-пайплайнах и тесное взаимодействие с экосистемой инструментов мониторинга и наблюдаемости. Написанный на языке Go с встроенной средой выполнения JavaScript (Goja), k6 сочетает высокую производительность компилируемого языка с удобством сценарного описания на JavaScript.

Сценарии в k6 описываются в виде JavaScript-модулей, экспортирующих функцию `default`, которая определяет поведение одного виртуального пользователя. Встроенный API предоставляет функции для выполнения HTTP-запросов, проверки ответов (`checks`), управления группами запросов, работы с метриками, паузами, Cookie-хранилищем и контекстом виртуального пользователя. Использование JavaScript обеспечивает низкий порог входа для разработчиков, знакомых с веб-технологиями.

Одним из главных конкурентных преимуществ k6 является его производительность. В отличие от потоково-ориентированных инструментов (JMeter) и greenlet-based решений (Locust), k6 реализует виртуальных пользователей как лёгкие горютины Go, что обеспечивает минимальное потребление памяти и процессора при максимальном числе одновременных пользователей. Это позволяет генерировать значительную нагрузку даже с одной машины среднего класса.

k6 поддерживает гибкое управление профилями нагрузки через концепцию `executors`: разработчик может задать постоянное число виртуальных пользователей, нарастающую нагрузку, фиксированное число итераций в секунду (`constant arrival rate`) или комбинированный сценарий с несколькими стадиями. `Thresholds` (пороговые значения) позволяют автоматически завершать тест с ошибкой, если метрики выходят за допустимые пределы, что делает k6 особенно удобным для использования в качестве gate-проверки в CI/CD.

Встроенная система метрик k6 предоставляет детальную статистику: перцентили времени ответа (`p(50)`, `p(90)`, `p(95)`, `p(99)`), пропускную способность, число ошибок, данные о соединениях и объёме трафика.

Результаты могут быть экспортированы в формат JSON, CSV или переданы в реальном времени во внешние системы через output-плагины: InfluxDB, Prometheus, Grafana Cloud, Datadog, StatsD и другие.

Отдельного внимания заслуживает облачная платформа Grafana Cloud k6 (ранее – k6 Cloud), предоставляющая управляемую инфраструктуру для запуска распределённых тестов, хранения истории результатов, построения интерактивных отчётов и сравнения запусков между собой. Это позволяет командам запускать масштабные тесты без необходимости управлять собственными агентами, хотя и требует использования платного облачного сервиса.

Вместе с тем k6 имеет ряд ограничений. Использование JavaScript как языка сценариев, несмотря на широкую известность, означает, что среда выполнения намеренно урезана: недоступны привычные Node.js API (fs, net, process и др.), что может создавать сложности при переиспользовании существующего кода. Поддержка встроенного браузерного тестирования появилась относительно недавно и остаётся в стадии развития.

Распределённое тестирование без использования облачной платформы Grafana потребует ручной настройки оркестрации агентов: k6 сам по себе не предоставляет встроенного механизма координации нескольких экземпляров в стиле master–worker. Для запуска в Kubernetes необходимо использование сторонних операторов (k6-operator), что добавляет зависимость от инфраструктуры и усложняет первоначальную настройку.

Таким образом, k6 представляет собой зрелый, высокопроизводительный инструмент, хорошо вписывающийся в современные DevOps-практики и сочетающий удобство JavaScript-сценариев с производительностью Go-рантайма. Его ограничения проявляются преимущественно при необходимости самостоятельного управления распределённой инфраструктурой агентов, что делает актуальной разработку специализированного агентского компонента, способного работать в связке с управляющей платформой.

1.3 Недостатки существующих решений

В ходе анализа современных инструментов нагрузочного тестирования было выявлено, что ни одно из рассмотренных решений не является универсальным и не может в полной мере удовлетворить все требования, предъявляемые к легкому, высокопроизводительному агенту, интегрированному с управляющей платформой. Ниже представлена таблица 1.1, обобщающая ключевые преимущества и недостатки трех рассмотренных инструментов: Apache JMeter, Locust и k6.

Таблица 1.1 – Преимущества и недостатки инструментов нагрузочного тестирования

Критерий	Apache JMeter	Locust	k6
Модель виртуальных пользователей	Поток на пользователя	Greenlet	Горутина
Потребление ресурсов	Высокое	Среднее	Низкое
Описание сценариев	XML	Python-код	JavaScript-код
Порог входа	Низкий	Средний	Низкий
Распределенный режим	Встроенный	Встроенный	Требует оркестратора
Интеграция с CI/CD	Ограниченная	Хорошая	Отличная
Встроенная отчетность	Развитая	Минималистичная	Подробная
Интеграция с мониторингом	Через плагины	Через плагины	Нативная
Поддержка протоколов	Широкая (HTTP, FTP, JDBC и др.)	HTTP, расширяемая	HTTP, Расширяемая
Декларативные YAML-сценарии	Нет	Нет	Нет
Лицензия	Apache 2.0	MIT	AGPL-3.0

Суммируя все вышесказанное, можно сделать однозначный вывод: на рынке отсутствует легкий, высокопроизводительный агент нагрузочного тестирования, который сочетал бы декларативное YAML-описание сценариев с поддержкой переменных и условных переходов, минимальное потребление ресурсов, нативную интеграцию с управляющей платформой и простоту развертывания в Docker- и Kubernetes-окружения.

2 ПОСТАНОВКА ЗАДАЧИ

Проведённый анализ современных инструментов нагрузочного тестирования показал, что несмотря на многообразие представленных на рынке решений, отсутствует лёгкий, специализированный агент, способный одновременно обеспечить декларативное описание сценариев, высокую производительность, нативную интеграцию с управляющей платформой и простоту развёртывания. Такие инструменты, как Apache JMeter, Locust и k6, хорошо решают отдельные задачи, однако не могут в полной мере охватить потребности команд, стремящихся к полному контролю над распределённой инфраструктурой нагрузочного тестирования и глубокой интеграции с собственной платформой управления тестами.

Цель проекта заключается в разработке лёгкого, высокопроизводительного агента нагрузочного тестирования, объединяющего лучшие свойства современных инструментов, но при этом лишённого их ключевых недостатков. Агент должен обеспечивать выполнение параметризованных HTTP-сценариев с поддержкой переменных, условных переходов и извлечения данных из ответов, эффективно использовать вычислительные ресурсы при высоком числе виртуальных пользователей, а также передавать метрики производительности в управляющую платформу в режиме реального времени.

Для реализации поставленной цели необходимо решить следующие задачи:

- 1) разработать формат описания тестовых сценариев на основе YAML с поддержкой переменных, подстановки значений, path-параметров, заголовков, тела запроса, задержек и условных переходов между шагами;
- 2) реализовать парсер и валидатор сценариев, обеспечивающий корректность входных данных до начала выполнения теста;
- 3) разработать исполнитель HTTP-запросов с поддержкой всех стандартных методов, управления Cookie-хранилищем, тайм-аутов и контекстной отмены;
- 4) реализовать модель виртуального пользователя, обеспечивающую изоляцию контекста, сохранение переменных между шагами сценария и выполнение условных переходов на основе HTTP-статуса ответа;
- 5) разработать планировщик нагрузки, поддерживающий постепенное нарастание числа виртуальных пользователей и корректное завершение теста по истечении заданной длительности или при получении сигнала остановки;
- 6) реализовать интерфейс командной строки для запуска агента, валидации сценариев в режиме dry-run и управления параметрами подключения к управляющей платформе.

2.1 Архитектура проектируемой системы

С учётом сформулированных выше задач и выявленных в первой главе недостатков существующих решений к проектируемой системе предъявляется ряд архитектурных требований. Система должна допускать горизонтальное масштабирование подсистемы генерации нагрузки, обеспечивать изоляцию отказов отдельных агентов от управляющей инфраструктуры, не накладывать ограничений на сетевую топологию развёртывания агентов и позволять независимое развитие подсистемы хранения и подсистемы исполнения. В качестве архитектурной модели, удовлетворяющей этим требованиям, целесообразно принять разделённую архитектуру «control plane – data plane», получившую широкое распространение в современных облачных платформах нагрузочного тестирования и доказавшую свою применимость при работе с большим числом распределённых исполнителей.

Согласно этой модели, система должна быть разделена на две функционально и развёртываемо независимые части. Управляющая платформа выступает в роли серверной части и должна сосредотачивать в себе задачи ведения проектов и пользователей, хранения сценариев, регистрации агентов, приёма результатов и формирования агрегированной отчётности; задачи генерации нагрузки на тестируемую систему на платформу не возлагаются. Агент нагрузочного тестирования должен представлять собой автономный исполняемый компонент, развёртываемый на произвольном числе машин, виртуальных серверов или контейнеров, и решать единственную задачу – генерировать HTTP-нагрузку согласно полученному от платформы сценарию с одновременной передачей измеренных метрик обратно на платформу. Несколько агентов должны иметь возможность работать параллельно, обслуживая независимые запуски и не оказывая друг на друга влияния.

К протоколу взаимодействия агента и платформы предъявляются следующие требования: применение общедоступного протокола HTTP с передачей данных в формате JSON для обеспечения совместимости и удобства отладки; инициация всех соединений со стороны агента для устранения необходимости в открытии входящих портов и допустимости развёртывания агентов за NAT, в частных подсетях и в кластерах; пассивная роль платформы в роли приёмника запросов. Совокупность этих требований определяет три самостоятельных цикла взаимодействия, реализация которых необходима в проектируемой системе:

- 1) цикл регистрации, выполняемый однократно при первом запуске агента, в рамках которого одноразовый enrollment-токен, выданный платформой при создании агента, обменивается на долговременный session-токен;

2) цикл heartbeat, поддерживающий присутствие агента в системе и обеспечивающий обновление в платформе сведений о его версии и времени последнего обращения;

3) цикл захвата запусков, основанный на механизме длинного опроса: агент периодически опрашивает платформу о наличии назначенного ему запуска, а платформа при отсутствии работы удерживает соединение в течение заданного окна и затем возвращает пустой ответ, после чего агент немедленно открывает новое соединение.

Помимо серверной части управляющая платформа должна включать клиентское веб-приложение, реализующее пользовательский интерфейс к функциям платформы. Веб-приложение должно поддерживать аутентификацию, управление проектами, агентами и тестовыми запусками, отображение результатов выполненных тестов, ошибок, возникших по ходу их выполнения, а также предоставлять визуальный редактор тестовых сценариев с возможностью графического описания шагов и условных переходов между ними, с сохранением значений в контекст виртуального пользователя, и автоматического импорта эндпоинтов из OpenAPI-спецификации тестируемого сервиса. Взаимодействие веб-приложения с серверной частью должно осуществляться исключительно по REST API. В рамках настоящей дипломной работы клиентское приложение рассматривается лишь как потребитель этого API и подробному проектированию не подлежит.

Схема взаимодействия компонентов проектируемой системы представлена на рисунке 2.1.

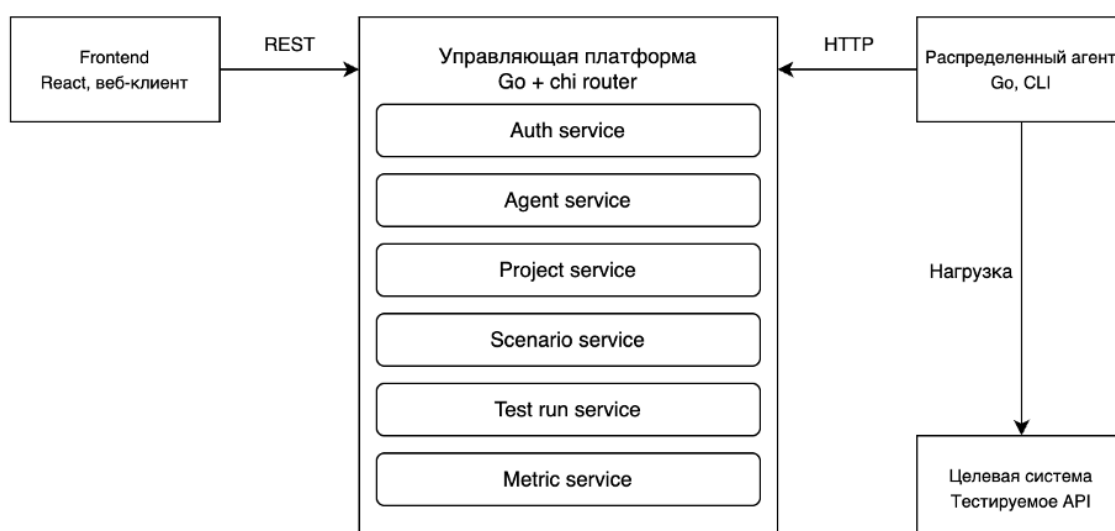


Рисунок 2.1 – Схема взаимодействия компонентов проектируемой системы

К основным компонентам системы и их назначению предъявляются следующие требования:

1) клиентское веб-приложение должно предоставлять пользователю интерфейс управления проектами, агентами и запусками тестов, визуальный редактор тестовых сценариев и средства просмотра результатов выполненных тестов; взаимодействие с серверной частью должно выполняться исключительно по REST API;

2) управляющая платформа должна быть реализована в виде серверной части с трёхслойной архитектурой `transport` → `service` → `repository`, обрабатывать запросы пользовательского интерфейса и распределённых агентов и хранить все сущности системы в единой встроенной СУБД;

3) агент нагрузочного тестирования должен представлять собой автономный исполняемый компонент, регистрирующийся в платформе по одноразовому `enrollment`-токену, поддерживающий своё присутствие в системе через цикл `heartbeat` и забирающий запуски через механизм длинного опроса;

4) тестируемая система рассматривается как внешний по отношению к проектируемой системе объект и в её состав не входит;

5) встроенная СУБД должна выступать в роли единого хранилища управляющей платформы, обеспечивая сохранение пользователей, организаций, проектов, агентов, сценариев, запусков и метрик в одном файле базы данных без необходимости развёртывания внешнего сервера.

Принятие описанной архитектурной модели обеспечит прямолинейное масштабирование подсистемы генерации нагрузки за счёт добавления новых агентов, исключит необходимость переноса бизнес-логики на сторону агента, обеспечит изоляцию подсистемы хранения от подсистемы исполнения и создаст условия для независимой разработки, версионирования и развёртывания обеих частей системы.

2.2 Варианты использования

На основании сформулированных задач и принятой архитектурной модели в проектируемой системе предусматривается два типа действующих лиц: пользователь и агент нагрузочного тестирования. Пользователь представляет собой инженера по тестированию или разработчика, взаимодействующего с системой через веб-интерфейс. Агент представляет собой автономный программный компонент, взаимодействующий с системой через REST API на правах отдельного действующего лица: после регистрации он самостоятельно опрашивает платформу о назначенных ему запусках и передаёт результаты. Такое разделение действующих лиц позволяет чётко разграничить зоны ответственности: пользователь сосредоточен на постановке задач тестирования, настройке сценариев и анализе результатов, тогда как агент берёт на себя непосредственное выполнение сценариев нагрузки и сбор

статистических данных. Подобная архитектура обеспечивает возможность горизонтального масштабирования за счёт одновременного подключения нескольких агентов к одной платформе и снимает с пользователя необходимость вручную управлять процессом выполнения нагрузочных тестов на стороне клиентской инфраструктуры.

Диаграмма вариантов использования, отражающая основные сценарии взаимодействия действующих лиц с проектируемой системой, представлена на рисунке 2.2.

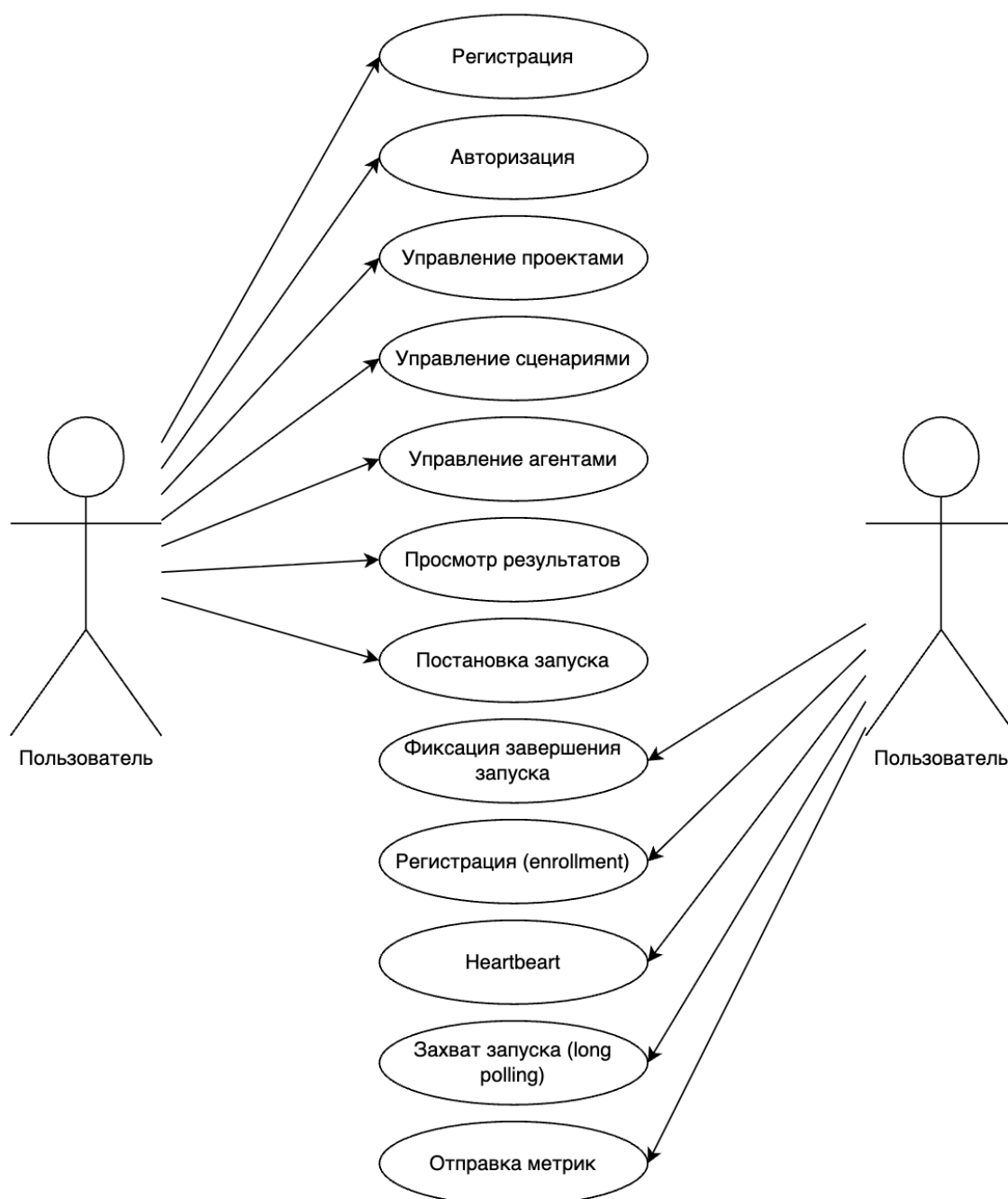


Рисунок 2.2 – Диаграмма вариантов использования

Перечень основных вариантов использования, которые должны быть реализованы в системе:

1) для пользователя:

- регистрация в системе и авторизация;
- создание проекта в рамках организации пользователя;
- создание, удаление и редактирование тестового сценария;
- создание агента и получение enrollment-токена для подключения;
- постановка запуска тестового сценария на выбранном агенте;
- просмотр статуса запуска и агрегированной сводки результатов;
- удаление агента;

2) для агента нагрузочного тестирования:

- регистрация в системе по одноразовому enrollment-токену и получение постоянного session-токена;
- поддержание присутствия в системе через периодические heartbeat-сообщения;
- захват очередного назначенного запуска через механизм длинного опроса;
- пакетная отправка метрик на платформу в процессе выполнения тестового сценария;
- фиксация финального статуса запуска и итоговой сводки по его завершении.

Подача задания агенту, фиксация финального состояния запуска и приём пакетных метрик должны образовывать согласованный цикл взаимодействия между пользователем и агентом, опосредованный управляющей платформой: пользователь не должен обращаться к агенту напрямую, а агент не должен иметь сведений о пользователе, инициировавшем запуск. Такое разграничение обеспечит изоляцию подсистем и упростит поддержку многопользовательской среды.

3 МЕТОДИЧЕСКИЕ ОСНОВЫ РАЗРАБОТКИ ПРИЛОЖЕНИЯ

3.1 Выбор и обоснование архитектуры агента

Разработка программного обеспечения для распределённого нагрузочного тестирования требует тщательного подхода к выбору архитектуры. Перед разработчиками встаёт принципиальный вопрос: реализовать монолитный инструмент, самостоятельно выполняющий все функции, или создать специализированный агент, работающий в связке с управляющей платформой. Это решение во многом определяет дальнейшую судьбу проекта, его масштабируемость и применимость в реальных инфраструктурах.

Традиционный монолитный подход к нагрузочному тестированию предполагает единое приложение, которое самостоятельно описывает сценарий, генерирует нагрузку, собирает метрики и отображает результаты. Именно по такому принципу построены классические инструменты – Apache JMeter и ранние версии Locust. Для небольших проектов такой подход обладает неоспоримыми преимуществами. Прежде всего, это простота установки и запуска: пользователь получает единый исполняемый файл, не требующий настройки взаимодействия с внешними системами. Все компоненты работают в едином процессе, что упрощает отладку и снижает вероятность ошибок на стыке подсистем.

Производительность монолитных решений в рамках одной машины также заслуживает внимания. Отсутствие необходимости в сетевом взаимодействии между компонентами позволяет добиться минимальных задержек при координации виртуальных пользователей и сборе результатов.

Однако монолитная архитектура не лишена существенных недостатков. По мере роста нагрузочных требований одна машина перестаёт справляться с генерацией достаточного числа виртуальных пользователей. Масштабирование монолита требует ручного запуска нескольких экземпляров с последующей ручной агрегацией результатов, что значительно усложняет эксплуатацию. Кроме того, тесная связь между подсистемой генерации нагрузки и подсистемой отображения результатов не позволяет эффективно разделить ответственность между компонентами и затрудняет интеграцию с внешними платформами управления тестами.

Альтернативой монолиту выступает разделённая архитектура, в которой управляющая платформа отвечает за хранение сценариев, планирование запусков и отображение результатов, а лёгкие агенты непосредственно генерируют нагрузку и передают метрики. Именно такой подход реализован в современных облачных платформах нагрузочного тестирования.

Главное преимущество этой архитектуры – исключительная масштабируемость. Для увеличения нагрузки достаточно запустить дополнительные экземпляры агента на новых машинах или в Kubernetes-кластере, не внося изменений в сценарий или управляющую систему. Агент может быть развёрнут в любой среде: на физическом сервере, виртуальной машине, в Docker-контейнере или в облаке, что обеспечивает максимальную гибкость в выборе инфраструктуры.

Специализация агента на единственной задаче – исполнении HTTP-сценариев – позволяет сосредоточить усилия на достижении максимальной производительности и минимального потребления ресурсов. Отсутствие графического интерфейса и аналитических подсистем делает агент лёгким и предсказуемым в поведении под нагрузкой.

Вместе с тем данная архитектура предъявляет более высокие требования к проектированию протокола взаимодействия между агентом и платформой, обеспечению согласованности результатов при параллельной работе нескольких агентов и корректной обработке сетевых сбоев.

Для реализации дипломного проекта была выбрана архитектура специализированного агента, что обусловлено рядом факторов. Во-первых, система изначально проектируется как компонент управляющей платформы, а значит, агент должен быть независимым, переносимым и легко горизонтально масштабируемым. Во-вторых, специализация на задаче исполнения HTTP-сценариев позволяет достичь максимальной производительности при минимальных требованиях к ресурсам.

Агентская архитектура позволит развивать систему по частям: совершенствовать логику исполнения сценариев, расширять поддерживаемые протоколы и обновлять механизм передачи метрик независимо от управляющей платформы.

3.2 Выбор средств разработки для реализации системы

3.2.1 Язык программирования Go

В качестве основного языка для реализации агента выбран Go (Golang). Этот язык был разработан компанией Google и на сегодняшний день широко используется при создании высокопроизводительных сетевых приложений, распределённых систем и инструментов DevOps-инфраструктуры. Go выбран за его простоту, лаконичность синтаксиса, встроенную поддержку конкурентности и отличную производительность при низком потреблении ресурсов. Дополнительным преимуществом является компиляция в единый статически слинкованный исполняемый файл без внешних зависимостей, что

упрощает распространение агента и его запуск на любой целевой платформе, а модель горутин обеспечивает эффективное моделирование большого числа одновременных виртуальных пользователей в пределах одного процесса.

Go предоставляет встроенные механизмы конкурентности – горутины (goroutines) и каналы (channels), – которые позволяют легко реализовать параллельное выполнение тысяч виртуальных пользователей без накладных расходов, характерных для традиционных потоков операционной системы. Горутина потребляет около 2–4 КБ памяти при запуске, тогда как системный поток требует как минимум 1 МБ, что делает Go идеальным выбором для агента, одновременно управляющего большим числом виртуальных пользователей.

Статическая типизация и компиляция в нативный машинный код обеспечивают высокую предсказуемость производительности и позволяют обнаруживать ошибки на этапе сборки, до запуска теста. Это особенно важно для агента, работающего в автономном режиме: ошибка в логике исполнения сценария должна быть выявлена как можно раньше. Компиляция в единый бинарный файл без внешних зависимостей упрощает упаковку агента в Docker-образ: в финальный контейнер достаточно поместить один исполняемый файл, без среды выполнения и системных библиотек. Это уменьшает размер образа и сокращает поверхность атаки, что важно для агента, разворачиваемого в кластере во множестве экземпляров.

Существенное практическое значение имеет также скорость компиляции Go. Сборка проекта объёмом в несколько десятков тысяч строк кода занимает единицы секунд, что обеспечивает быстрый цикл обратной связи при разработке агента и не накладывает значимых задержек на CI/CD-конвейер. Это особенно важно для проектов, в которых каждый коммит сопровождается автоматической сборкой бинарного файла, сборкой Docker-образа и публикацией артефактов в реестр.

Дополнительным преимуществом является встроенный набор инструментов разработки, поставляемый вместе с компилятором Go. Утилиты `go fmt`, `go vet`, `go test` и `go mod` обеспечивают единое форматирование кода, статический анализ, выполнение модульных и интеграционных тестов, а также управление зависимостями без необходимости подключения сторонних инструментов сборки. Унификация инструментария существенно снижает порог входа новых участников в проект и упрощает сопровождение кодовой базы на длительном промежутке времени.

Также значимым фактором при выборе Go стало наличие зрелой стандартной библиотеки, покрывающей большинство потребностей агента без привлечения сторонних зависимостей: работа с HTTP, обработка контекстов отмены, работа с сигналами операционной системы и многое другое. Таким

образом, выбор Go обусловлен его техническими преимуществами в области конкурентности и производительности, соответствием требованиям агентской архитектуры и широкой применимостью в экосистеме инструментов нагрузочного тестирования.

3.2.2 Библиотека командной строки Cobra

Для реализации интерфейса командной строки агента была выбрана библиотека Cobra версии 1.10. Cobra является де-факто стандартом для разработки CLI-приложений на языке Go и используется в таких широко известных проектах, как Kubernetes, Docker, Hugo и ряде других.

Cobra предоставляет иерархическую модель команд, при которой каждая команда может содержать собственный набор флагов и вложенные подкоманды. Это позволяет логично структурировать интерфейс агента: команда `connect` реализует регистрацию агента на управляющей платформе и поддержание постоянного подключения через механизм `heartbeat`, команда `run` обеспечивает непосредственное выполнение тестового сценария из файла, а команда `version` выводит сведения о текущей версии сборки. Иерархическая модель делает интерфейс агента интуитивно понятным и легко расширяемым при добавлении новых команд.

Cobra тесно интегрируется с библиотекой `pflag`, обеспечивающей разбор флагов командной строки в стиле POSIX/GNU. Флаги могут быть привязаны к переменным среды через механизм `envOrDefault`, что позволяет управлять конфигурацией агента как через командную строку, так и через переменные окружения – стандартный подход для контейнерных развёртываний. Например, URL управляющей платформы может быть задан флагом `--platform-url` или переменной среды `LOADFORGE_PLATFORM_URL`.

Cobra автоматически генерирует справочные сообщения для каждой команды и подкоманды, что снижает трудозатраты на написание документации. Обработка ошибок выполняется единообразно: при некорректных аргументах фреймворк выводит подсказку с описанием допустимого синтаксиса. Таким образом, Cobra обеспечивает профессиональный и удобный в использовании интерфейс командной строки при минимальных затратах на его реализацию.

3.2.3 Стандартная библиотека `net/http`

Для выполнения HTTP-запросов в рамках тестовых сценариев задействована стандартная библиотека `net/http` языка Go. Данная библиотека входит в состав стандартной поставки языка, не требует сторонних

зависимостей и предоставляет полноценный HTTP-клиент с поддержкой всех стандартных методов (GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS), управления заголовками, Cookie-хранилищем, перенаправлениями и тайм-аутами.

Ключевым преимуществом `net/http` с точки зрения нагрузочного агента является нативная поддержка контекстов отмены. Каждый HTTP-запрос может быть привязан к `context.Context`, что позволяет корректно прерывать выполнение по истечении тайм-аута или при получении сигнала остановки теста, не оставляя «подвисших» горутин. Это критически важно при управлении тысячами параллельных виртуальных пользователей: утечка горутин в агенте нагрузочного тестирования напрямую сказывается на достоверности результатов.

Библиотека также поддерживает настройку пула соединений через структуру `http.Transport`: параметры `MaxIdleConnsPerHost`, `DialTimeout`, `TLSHandshakeTimeout` и другие позволяют тонко настраивать поведение клиента под конкретный сценарий нагрузочного теста. Использование `net/http` исключает необходимость введения сторонней зависимости для ключевой функции агента и гарантирует предсказуемость поведения во всех поддерживаемых операционных системах.

3.2.4 Библиотека `gorpkg.in/yaml.v3` для разбора сценариев

Для разбора файлов тестовых сценариев применяется библиотека `gorpkg.in/yaml.v3` – наиболее зрелая и широко используемая YAML-библиотека экосистемы Go. YAML был выбран в качестве формата описания сценариев, поскольку он хорошо читается человеком, поддерживает иерархические структуры данных и широко применяется в DevOps-инструментах.

Библиотека `yaml.v3` обеспечивает маршалинг и анмаршалинг YAML-документов в строго типизированные Go-структуры. Это позволяет определить схему сценария в виде набора `struct`-типов с тегами `yaml`, после чего библиотека автоматически валидирует типы полей при разборе файла. Несоответствие типов или отсутствие обязательных полей обнаруживается на этапе загрузки сценария – до начала нагрузочного теста.

`yaml.v3` поддерживает якоря и алиасы YAML, что позволяет переиспользовать фрагменты конфигурации внутри одного файла сценария. Библиотека корректно обрабатывает многострочные строки, что важно для сценариев с телом HTTP-запроса в формате JSON или XML. Кроме того, `yaml.v3` допускает определение собственных правил разбора для отдельных типов, что в проекте используется для полей задержки между шагами, задаваемых в человекочитаемом виде (например, `2s` или `500ms`). Таким

образом, `gorpkg.in/yaml.v3` обеспечивает надёжную и удобную основу для разбора декларативных сценариев нагрузочного тестирования.

3.2.5 Библиотека `gjson` для извлечения данных из ответов

Для извлечения значений из JSON-ответов сервера применяется библиотека `github.com/tidwall/gjson` версии 1.18. Данная библиотека реализует высокопроизводительный поиск по JSON-документу с использованием `path-выражений`, не требуя предварительного полного разбора документа в структуру данных Go. Это особенно важно для агента нагрузочного тестирования, где тысячи виртуальных пользователей параллельно обрабатывают ответы сервера.

`gjson` поддерживает точечную нотацию для навигации по вложенным объектам (`user.id`, `token.access`), индексирование массивов (`items.0.name`), а также операторы для поиска по значению и агрегации. Это позволяет реализовать в сценарии типичный паттерн нагрузочного тестирования: выполнить запрос аутентификации, извлечь из ответа JWT-токен через `gjson-выражение` и передать его в заголовок последующих запросов в рамках той же сессии виртуального пользователя.

Производительность `gjson` подтверждается бенчмарками: библиотека работает без аллокаций памяти для большинства базовых запросов, что делает её использование в горячем пути выполнения сценария безопасным с точки зрения нагрузки на сборщик мусора Go. Таким образом, `gjson` обеспечивает гибкое и эффективное извлечение данных из JSON-ответов, являясь ключевым инструментом для реализации сценариев с передачей состояния между шагами.

3.2.6 Библиотека `kin-openapi` для валидации сценариев

Библиотека `kin-openapi` предоставляет инструменты для загрузки и разбора спецификаций OpenAPI 3.x в виде типизированной объектной модели Go, а также средства валидации как самих спецификаций, так и HTTP-запросов и ответов относительно заданной схемы. Перед использованием спецификации агент выполняет её валидацию средствами библиотеки: некорректный или не соответствующий стандарту документ отклоняется уже на этапе импорта, до начала генерации сценариев. Из разобранной модели извлекается полный набор операций – HTTP-метод, путь, теги, идентификатор и описание операции, а также адреса серверов, – который служит исходным материалом для автоматического построения тестовых сценариев; дополнительно поддерживается фильтрация операций по тегам, позволяющая

ограничить тестирование заданным подмножеством API. Наличие в библиотеке средств валидации запросов и ответов архитектурно открывает возможность дальнейшей реализации режима *schema-validation*, при котором агент в ходе нагрузочного теста сможет проверять соответствие ответов сервера контрактам, описанным в OpenAPI-спецификации.

Таким образом, *kin-openapi* расширяет возможности агента за пределы простого нагрузочного тестирования: совмещение генерации нагрузки с контрактным тестированием позволит выявлять как деградацию производительности, так и нарушения API-контракта в едином тестовом прогоне.

3.2.7 Docker и контейнеризация

Для обеспечения переносимости и удобства развёртывания агента применяются технологии контейнеризации на базе Docker. Docker позволяет упаковать агент вместе со всеми зависимостями в единый образ, гарантируя идентичное поведение на любой платформе – будь то локальная машина разработчика, виртуальный сервер или Kubernetes-кластер облачного провайдера. Это особенно важно для агента нагрузочного тестирования, который может разворачиваться на десятках и сотнях машин одновременно.

Поскольку Go компилируется в статически связанные бинарные файлы без внешних зависимостей, образ Docker на базе *scratch* или *alpine* получается минимальным по размеру. Это ускоряет загрузку образа при горизонтальном масштабировании и снижает поверхность атаки с точки зрения безопасности. Отсутствие CGO-зависимостей в использованных библиотеках (в частности, modernc.org/sqlite в платформе) позволяет собирать образы без установки дополнительных инструментов сборки.

Контейнеризация также обеспечивает изоляцию агента от хостовой системы: агент не имеет доступа к файловой системе, переменным среды и сетевым интерфейсам хоста за пределами явно заданных. Конфигурация агента – URL платформы, токен зачисления, параметры нагрузки – передаётся через переменные среды контейнера, что соответствует двенадцатифакторной методологии разработки *cloud-native* приложений. Таким образом, Docker обеспечивает надёжную, воспроизводимую и безопасную среду выполнения для агента нагрузочного тестирования при любом масштабе развёртывания.

4 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СИСТЕМЫ

4.1 Структура проекта

В процессе разработки распределённой системы нагрузочного тестирования мной была организована чёткая и модульная структура проекта, представленная на рисунке 4.1, позволяющая удобно сопровождать систему, расширять её функциональность и обеспечивать изоляцию компонентов.

loadforge-agent/	репозиторий агента
├─ cmd/agent/	точка входа, main
└─ internal/	внутренние пакеты агента
├─ cli/	CLI на cobra: run, connect, version
├─ scenario/	YAML, парсер, переменные
├─ executor/	HTTP-исполнитель поверх net/http
├─ runner/	VirtualUser, Stats, планировщик
└─ platform/	HTTP-клиент управляющей платформы
loadforge-platform/	репозиторий управляющей платформы
├─ backend/	серверная часть на Go
└─ cmd/server/	точка входа сервера
└─ internal/	внутренние пакеты бэкенда
└─ domain/	сущности предметной области
└─ auth/	JWT, bcrypt
└─ transport/http/	chi, handlers, middleware, DTO
└─ service/	бизнес-логика (6 сервисов)
└─ repository/sqlite/	sqlc, доступ к данным
└─ migrations/	goose-миграции
└─ frontend/	React SPA, потребитель REST API
deploy/	Dockerfile + docker-compose.yml

Рисунок 4.1 – Структура проекта

Проект реализован с применением разделённой архитектуры и разделён на два самостоятельных репозитория: репозиторий агента и репозиторий управляющей платформы, последний из которых содержит как серверную, так и клиентскую части.

Для удобства разработки и развёртывания все компоненты системы структурированы в едином корневом репозитории каждого проекта, в котором каждая часть представлена отдельной папкой со своей логикой, зависимостями и конфигурацией.

В основе системы лежит набор подсистем, каждая из которых представляет собой автономный, слабо связанный с остальными блок с чётко выделенной предметной областью и функциональностью. Такое модульное построение упрощает независимую разработку, тестирование и сопровождение отдельных частей системы, а также позволяет развивать их без риска затронуть работу смежных компонентов. Для удобства разработки и поддержки проект организован по следующим основным директориям и модулям:

- 1) `loadforge-agent/` – репозиторий распределённого агента нагрузочного тестирования;
- 2) `cmd/agent/` – точка входа агента, содержащая функцию `main` и запуск корневой команды;
- 3) `internal/cli/` – реализация командно-строчного интерфейса агента на основе библиотеки `cobra`: команды `run`, `connect`, `version`;
- 4) `internal/scenario/` – модели данных сценария, YAML-парсер, валидатор и подсистема подстановки переменных;
- 5) `internal/executor/` – низкоуровневый исполнитель HTTP-запросов поверх стандартной библиотеки `net/http`;
- 6) `internal/runner/` – ядро нагрузки: планировщик виртуальных пользователей, модель `VirtualUser` и потокобезопасный агрегатор статистики `Stats`;
- 7) `internal/platform/` – HTTP-клиент для взаимодействия с управляющей платформой и хранение сессионного состояния агента;
- 8) `loadforge-platform/` – репозиторий управляющей платформы, объединяющий серверную и клиентскую части;
- 9) `backend/cmd/server/` – точка входа сервера управляющей платформы;
- 10) `backend/internal/domain/` – чистые сущности предметной области без зависимостей от инфраструктуры;
- 11) `backend/internal/auth/` – JWT-подсистема и работа с bcrypt-хешами паролей;
- 12) `backend/internal/service/` – слой бизнес-логики: `AuthService`, `AgentService`, `ProjectService`, `ScenarioService`, `TestRunService`, `MetricService`;
- 13) `backend/internal/repository/sqlite/` – слой доступа к данным на основе `sqlc`-сгенерированного кода для встроенной СУБД `SQLite`;
- 14) `backend/internal/transport/http/` – HTTP-транспорт на основе маршрутизатора `chi`: обработчики, `middleware` и DTO-структуры;
- 15) `backend/migrations/` – миграции схемы базы данных в формате `goose`;
- 16) `frontend/` – клиентское одностраничное приложение на `React` (в данной работе рассматривается лишь как потребитель REST API серверной части);

17) `deploy/` – конфигурация контейнеризации, включающая `Dockerfile` для каждого компонента и `docker-compose.yml` для запуска всей системы.

4.2 Обоснование выбранной схемы контейнеризации компонентов

На начальном этапе проектирования рассматривался вариант полной контейнеризации системы – размещение всех её компонентов в рамках единого Docker-окружения. Основным мотивом такого решения являлась сетевая изоляция: требовалось гарантировать, что трафик агента и обрабатываемые им данные не покидают контролируемый периметр и не передаются во внешнюю сеть. Дополнительным преимуществом подхода считалась воспроизводимость окружения – возможность развернуть всю систему единообразно, одной командой, независимо от конфигурации хост-машины.

В ходе проработки концепции были выявлены два ограничения, вступающие в противоречие с заявленными целями.

Размещение всего технологического стека внутри одного контейнера приводит к высоким накладным расходам. Компоненты вынуждены конкурировать за вычислительные ресурсы (CPU, оперативная память) в пределах лимитов одного изолированного контейнера, при этом отсутствует горизонтальное распределение нагрузки, а вложенная изоляция ресурсов дополнительно значительно снижает эффективность их использования. Поскольку агент является наиболее ресурсоёмким компонентом и во многом определяет общую производительность системы, подобная деградация задержек и пропускной способности тестируемого приложения была признана неприемлемой.

Целевой средой развёртывания системы выступает кластер, развёрнутый в виде отладочного стенда, в котором функционирует горизонтальное автоматическое масштабирование (`autoscaling`) компонентов. Попытка воссоздать всю кластерную инфраструктуру внутри одного контейнера приводит к ряду принципиальных проблем:

1) утрачивается возможность проверки автомасштабирования: поведение механизма добавления и удаления реплик невозможно достоверно протестировать в пределах единственного контейнера;

2) изменяется топология системы относительно промышленного окружения, вследствие чего отладочная среда перестаёт быть репрезентативной, а результаты тестирования – некорректными;

3) фактически дублируется уже существующая инфраструктура: вместо использования готового, развёрнутого стенда она пересобирается заново в эфемерном контейнере, что не даёт выигрыша ни по одному из значимых критериев.

Сравнение вариантов схемы представлены в таблице 4.1

Таблица 4.1 – Сравнение вариантов схемы контейнеризации

Критерий	Полная контейнеризация (весь стек в одном контейнере)	Контейнеризация только агента
Изоляция сетевого трафика	Достигается	Достигается (средствами оркестратора)
Производительность	Низкая (конкуренция за ресурсы, отсутствие горизонтального масштабирования)	Высокая (нагрузка распределяется по кластеру)
Проверка автомасштабирования	Невозможна	Возможна
Репрезентативность системы	Низкая (топология отличается от целевой)	Высокая (топология совпадает с целевой)
Переиспользование инфраструктуры	Отсутствует (стенд собирается заново)	Присутствует (используется уже готовый стенд)

Таким образом, переход от полной контейнеризации к контейнеризации одного лишь агента представляет собой осознанный инженерный компромисс. Незначительная часть исходного замысла – развёртывание всей системы «одной командой» – была принесена в жертву ради сохранения репрезентативности отлаженной среды, возможности тестирования автомасштабирования и приемлемой производительности. При этом ключевая задача – изоляция сетевого трафика – была решена другими, более подходящими для целевой архитектуры средствами.

4.3 Описание компонентов агента

Распределённый агент нагрузочного тестирования реализован на языке Go и организован как набор внутренних пакетов, каждый из которых решает обособленную задачу в рамках единого процесса. Точкой входа агента служит пакет `cmd/agent`, делегирующий управление слою командной строки. Бизнес-логика распределена между пакетами `internal/cli`, `internal/scenario`,

internal/executor, internal/runner и internal/platform, что обеспечивает их независимую разработку, тестирование и эволюцию.

4.3.1 Командно-строчный интерфейс агента

Командно-строчный интерфейс агента реализован в пакете internal/cli поверх библиотеки cobra и предоставляет три основные команды: run, connect и version. Команда run принимает в качестве позиционного аргумента путь к YAML-файлу сценария и выполняет его в автономном режиме без обращения к управляющей платформе, что используется для локальной отладки сценариев. Флаг --dry-run позволяет ограничить выполнение этапом разбора и валидации сценария, не порождая фактической нагрузки на тестируемую систему.

Команда connect переводит агент в режим постоянного взаимодействия с управляющей платформой. При первом запуске агенту передаётся одноразовый enrollment-токен через флаг --enrollment-token или переменную среды LF_ENROLLMENT_TOKEN; полученный в обмен на него session-токен сохраняется в файл состояния (по умолчанию \$HOME/.loadforge/agent.json) с правами доступа 0600 и используется при последующих запусках без повторной регистрации. После успешной регистрации агент запускает параллельно два цикла: цикл heartbeat с настраиваемым через флаг --heartbeat-interval интервалом (по умолчанию 30 секунд) и цикл захвата запусков на основе длинного опроса. Команда version выводит сведения о версии сборки, идентификаторе git-коммита и дате компиляции, встраиваемых в исполняемый файл через флаги компоновщика -ldflags на этапе сборки.

Глобальные флаги, общие для всех команд, включают --platform-url для указания адреса управляющей платформы, --api-key для передачи аутентификационного материала и --verbose для включения подробного логирования.

4.3.2 Подсистема описания тестовых сценариев

Подсистема описания тестовых сценариев реализована в пакете internal/scenario и отвечает за разбор YAML-файлов в типизированную объектную модель, валидацию структуры сценария и подстановку переменных в шаги перед их выполнением. Центральной структурой данных является Scenario, содержащая имя теста, базовый URL тестируемой системы, число виртуальных пользователей, длительность теста в секундах, набор глобальных переменных и упорядоченный список шагов. Каждый шаг описывает отдельный HTTP-запрос к тестируемой системе.

Каждый шаг описывается структурой Step и содержит строку запроса в формате «МЕТОД /путь», набор тегов для группировки, заголовки, параметры строки запроса, path-параметры, тело запроса в произвольной структуре, задержку перед выполнением, инструкции извлечения значений из тела ответа на основе gjson-выражений (поле save_to_context) и список условных переходов к последующим шагам. Условный переход описывается структурой NextStep, целью которой выступает либо конкретный шаг по полю request, либо группа шагов с заданным тегом по полю tag со стратегией выбора random или sequential; выбор перехода основан на сравнении HTTP-статуса полученного ответа с заданным шаблоном (точное значение либо маска вида 2xx, 4xx).

Валидация сценария выполняется компонентом Parser сразу после загрузки YAML-файла и обеспечивает проверку обязательности базовых полей, корректности формата строки запроса (метод из числа разрешённых, путь, начинающийся с косой черты), допустимости значений задержек (не более десяти минут), длительности теста (не более одного года), уникальности шагов в рамках сценария, целостности ссылок между шагами в переходах и корректности форматов извлечения и сопоставления значений. Подсистема подстановки переменных, реализованная в компоненте Substitutor, применяет к шагу карту значений, объединяющую глобальные переменные сценария и контекст конкретного виртуального пользователя, и выполняет замену плейсхолдеров формата \${имя_переменной} в URL, заголовках, параметрах строки запроса, path-параметрах и теле запроса. Для тел JSON предусмотрено JSON-экранирование значений, что позволяет корректно подставлять строки, содержащие двойные кавычки, обратные слэши и переводы строк.

4.3.3 Низкоуровневый исполнитель HTTP-запросов

Низкоуровневый исполнитель HTTP-запросов реализован в пакете internal/executor поверх стандартной библиотеки net/http и предоставляет компонент Executor, инкапсулирующий HTTP-клиент с настроенным пулом соединений и Cookie-хранилищем (http.CookieJar). Каждый экземпляр виртуального пользователя владеет собственным экземпляром исполнителя, что обеспечивает изоляцию Cookie-состояния и сессионных заголовков между виртуальными пользователями: токен авторизации, полученный одним пользователем, не утечёт в запросы другого.

Структура Request описывает HTTP-запрос произвольным методом (поддерживаются GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, TRACE) с заданным URL, заголовками, телом и индивидуальным тайм-аутом. Метод Execute принимает контекст context.Context, что обеспечивает

корректное прерывание запроса по сигналу остановки теста или по истечении общего срока действия запуска. Возвращаемая структура `Response` содержит код ответа, текстовый статус, заголовки, тело и продолжительность выполнения запроса, измеренную с миллисекундной точностью. Общий тайм-аут HTTP-клиента по умолчанию составляет 30 секунд и может быть переопределён через индивидуальный тайм-аут запроса.

4.3.4 Ядро генерации нагрузки

Ядро генерации нагрузки реализовано в пакете `internal/runner` и включает три ключевых компонента: планировщик `Runner`, модель виртуального пользователя `VirtualUser` и потокобезопасный агрегатор статистики `Stats`. Планировщик создаёт заданное в сценарии число виртуальных пользователей, постепенно вводя их в работу в течение первых десяти процентов общей длительности теста, что соответствует стандартной модели нарастающей нагрузки (`ramp-up`). Каждый виртуальный пользователь исполняется в отдельной горутине и в цикле выполняет шаги сценария до получения сигнала об остановке от контекста.

Модель `VirtualUser` хранит собственную карту переменных, защищённую мьютексом, и счётчик последовательности выбора тегов для воспроизводимой ротации шагов по стратегии `sequential`. Выполнение одного шага состоит из подстановки переменных в шаблонные поля шага, формирования HTTP-запроса с применением `path`-параметров, заголовков, параметров строки запроса и тела, фактической отправки запроса через исполнитель, измерения задержки ответа, извлечения значений из тела ответа в контекст пользователя при помощи `gjson`-выражений и выбора следующего шага в соответствии с правилами условных переходов. При отсутствии явных переходов из текущего шага следующий шаг берётся из плоского порядка сценария.

Агрегатор статистики `Stats` собирает счётчики на горячем пути выполнения: общее число запросов, число ошибок, суммарную, минимальную и максимальную задержку, объём принятого трафика, а также аналогичные показатели в разбивке по конкретным эндпоинтам. Глобальные счётчики обновляются неблокирующим способом через атомарные операции пакета `sync/atomic`, что обеспечивает приемлемую производительность при темпе порядка тысячи запросов в секунду на ядро. Структуры с разбивкой по эндпоинтам и кольцевой буфер последних неуспешных запросов (размером сто записей) защищены единым мьютексом, что приемлемо при характерной нагрузке агента. По завершении теста агрегатор формирует структуру `Snapshot`, передаваемую в управляющую платформу в качестве итоговой

сводки запуска: она включает общие показатели, постатейную статистику по эндпоинтам и список последних зафиксированных ошибок.

4.3.5 Клиент управляющей платформы

Клиент управляющей платформы реализован в пакете `internal/platform` и инкапсулирует протокол взаимодействия агента с REST API серверной части. Компонент `Client` предоставляет методы `Enroll`, `Heartbeat`, `ClaimNextRun`, `FinishRun` и `IngestMetrics`, каждый из которых соответствует одному из агентских эндпоинтов платформы. Метод `Enroll` обменивает одноразовый `enrollment`-токен на постоянный `session`-токен; все последующие методы передают полученный `session`-токен в заголовке `Authorization` формата `Bearer <token>`.

Метод `ClaimNextRun` реализует длинный опрос с указанием максимального времени удержания соединения через параметр строки запроса `wait_seconds` (по умолчанию 25 секунд) и интерпретирует ответ платформы со статусом 204 как «работы нет в данном окне», возвращая нулевое значение и предписывая агенту немедленно повторить запрос. Метод `IngestMetrics` отправляет накопленные за фиксированный интервал точки метрик пакетом, что снижает число сетевых обращений к платформе и нагрузку на её подсистему записи. Метод `FinishRun` использует независимый от основного контекста таймаут продолжительностью десять секунд, благодаря чему сведения о финальном состоянии запуска успевают доставиться даже при штатном завершении агента.

Постоянное состояние агента сохраняется в файл состояния в формате JSON через подсистему `State`, что обеспечивает корректное переподключение агента после перезапуска без повторной выдачи `enrollment`-токена. Общий тайм-аут HTTP-клиента установлен в 60 секунд, что покрывает максимальную длительность длинного опроса с запасом.

4.4 Описание сервисов управляющей платформы

Серверная часть управляющей платформы реализована на языке Go 1.26 и построена по принципу трёхслойной архитектуры: слой транспорта принимает HTTP-запросы и выполняет десериализацию входных данных, слой бизнес-логики реализует прикладные операции и проверки прав доступа, слой доступа к данным инкапсулирует взаимодействие с СУБД через типизированный код, сгенерированный утилитой `sqlc`. Доменные сущности изолированы в пакете `internal/domain` и не имеют зависимостей от инфраструктуры, что обеспечивает их переиспользование между слоями.

Взаимодействие со слоем доступа к данным организовано через интерфейсы репозитория, что позволяет заменить используемую СУБД без изменения бизнес-логики и транспорта. Ниже описаны основные сервисы прикладного слоя.

4.4.1 Авторизационный сервис (AuthService)

AuthService отвечает за регистрацию пользователей, авторизацию по электронной почте и паролю, выпуск пар access-токенов и refresh-токенов, ротацию refresh-токенов и завершение сеансов. При регистрации сервис создаёт пользователя, автоматически порождает для него организацию и членство в ней с ролью владельца, что обеспечивает корректный старт работы в многопользовательской среде. Пароли пользователей хешируются библиотекой bcrypt; refresh-токены хранятся в базе данных в виде хешей, что предотвращает их раскрытие при компрометации содержимого СУБД.

При успешной авторизации сервис формирует JWT-токен подписи HS256, содержащий идентификатор пользователя и срок действия. Все защищённые пользовательские эндпоинты проходят через middleware RequireAuth, проверяющее валидность access-токена; недействительные или просроченные токены порождают стандартизованный ответ со статусом 401. Метод Refresh применяет ротацию: предъявленный refresh-токен помечается отозванным, а взамен выпускается новая пара access- и refresh-токенов, что ограничивает срок жизни любого скомпрометированного refresh-токена временем до его очередного использования.

4.4.2 Управление проектами (ProjectService)

ProjectService реализует управление проектами в рамках организации пользователя: создание, чтение, обновление, удаление и страничный вывод списка. Проект выступает контейнером для тестовых сценариев и не имеет прямых связей с агентами и запусками, что упрощает миграцию сценариев между проектами при необходимости. Перед выполнением каждой операции сервис проверяет членство пользователя в организации, к которой относится проект, что обеспечивает изоляцию данных между организациями в многопользовательской среде.

4.4.3 Аутентификация агентов на платформе (AgentService)

AgentService отвечает за регистрацию агентов в системе, выпуск enrollment-токенов и session-токенов, обработку heartbeat-сообщений и

аутентификацию агента по session-токену при каждом обращении к агентским эндпоинтам. При создании агента сервис формирует случайный enrollment-токен длиной 32 байта, сохраняет в базе данных только его SHA-256-хеш и возвращает оригинальное значение пользователю однократно – в дальнейшем оригинал в системе не существует.

В момент первого подключения агент обменивает enrollment-токен на session-токен через метод Enroll: сервис проверяет соответствие хешей, генерирует новый session-токен и сохраняет в базе данных его хеш, помечая агента как зарегистрированного через установку поля enrolled_at. Повторное использование enrollment-токена после успешной регистрации отклоняется. Метод Heartbeat обновляет в записи агента поле времени последнего обращения и сведения о версии исполняемого файла, что используется для отображения статуса агента в пользовательском интерфейсе. Метод AuthenticateByToken применяется одноимённым middleware RequireAgent на каждом агентском запросе и обеспечивает поиск агента по хешу предъявленного session-токена.

4.4.4 Управление тестовыми сценариями (ScenarioService)

ScenarioService обеспечивает управление тестовыми сценариями в рамках проекта: создание, чтение, обновление, удаление и получение списка. Сценарий хранится в каноническом JSON-представлении в поле config, что позволяет точно восстанавливать состояние графического редактора при последующем открытии. При постановке запуска компонент internal/scenario управляющей платформы преобразует это JSON-представление в YAML-формат, ожидаемый агентом, на стороне платформы, что освобождает клиентское приложение от необходимости синхронизироваться с агентом по формату сценария и гарантирует единую точку компиляции. Все операции выполняются с проверкой прав доступа через цепочку «сценарий → проект → организация → членство пользователя». Размер JSON-представления ограничен с целью защиты от непреднамеренного раздувания одного документа сценария.

4.4.5 Реализация тестового сценария (TestRunService)

TestRunService реализует жизненный цикл запуска тестового сценария на агенте. При создании запуска сервис проверяет совпадение организаций сценария и агента, фиксирует запуск в базе данных со статусом «pending» и публикует уведомление через внутренний компонент RunBus, что приводит к мгновенному пробуждению агента, ожидающего работу через длинный опрос.

Метод `ClaimNextForAgent` репозиторного слоя обеспечивает атомарный захват ожидающего запуска через единый SQL-оператор `UPDATE` с условием на текущий статус: благодаря этому даже при одновременном опросе одного и того же запуска двумя экземплярами агента он будет переведён в состояние «`running`» только в одном из них.

Длинный опрос на стороне платформы реализован методом `ClaimNextWait` через подписку на канал `RunBus` с окном ожидания до 25 секунд: при отсутствии готовых запусков сервис удерживает соединение до появления новой публикации либо до истечения окна, после чего отвечает агенту статусом 204. Компонент `RunBus` реализован в виде карты буферизованных каналов «один на агента» с буфером единицы, что гарантирует, что уведомление не теряется и при отсутствии слушателя дождётся следующего ожидающего вызова. Метод `Finish` фиксирует финальное состояние запуска: статус из числа «`completed`», «`failed`» или «`cancelled`», сводку результатов в формате JSON и при необходимости описание ошибки. Метод `VerifyRunForAgent` применяется в качестве вспомогательной проверки перед приёмом метрик и подтверждает, что запуск действительно принадлежит обращающемуся агенту и находится в активном состоянии.

4.4.6 Метрики (`MetricService`)

`MetricService` отвечает за приём пакетных метрик от агентов и формирование агрегированных сводок по конкретному запуску. Перед каждой вставкой сервис верифицирует, что обращающийся агент действительно владеет указанным запуском (через `TestRunService.VerifyRunForAgent`), что предотвращает попытку поддельной отправки результатов произвольным агентом. Метрики записываются в базу данных пакетом размером до пятисот точек за один запрос; превышение этого порога отклоняется как некорректный ввод. Метод `Summary` выполняет агрегированный SQL-запрос по таблице метрик и возвращает общее число запросов, число ошибочных запросов, среднюю, минимальную и максимальную задержку для конкретного запуска.

4.5 Интерфейсы и взаимодействия

Все компоненты системы взаимодействуют между собой по протоколу HTTP с передачей данных в формате JSON. Маршрутизация HTTP-запросов реализована поверх библиотеки `chi` и сгруппирована по двум корням: пользовательскому (`/api/v1/...` с аутентификацией через JWT-токен в заголовке `Authorization`) и агентскому (`/api/v1/agent/...` с аутентификацией через `session-`

токен). Защищённые эндпоинты проходят через одно из двух middleware (RequireAuth для пользователей, RequireAgent для агентов), что централизует логику аутентификации и упрощает её модификацию. Эндпоинт регистрации агента (/agent/enroll) является публичным, поскольку именно он обменивает enrollment-токен на постоянный session-токен.

4.5.1 Эндпоинты аутентификации

Эндпоинты аутентификации обеспечивают регистрацию пользователей, выпуск пар access– и refresh-токенов, ротацию refresh-токенов и завершение сеансов. Все эндпоинты этой группы являются публичными – middleware RequireAuth к ним не применяется, поскольку именно они отвечают за получение материалов аутентификации, необходимых для доступа к остальным защищённым ресурсам системы. При неуспешной аутентификации обработчики возвращают единообразную ошибку вне зависимости от причины отказа, что исключает возможность определения существования учётной записи по ответу сервера

В таблице 4.2 представлены HTTP-эндпоинты подсистемы аутентификации.

Таблица 4.2 – HTTP-эндпоинты подсистемы аутентификации

HTTP-метод	URI	Назначение запроса
1	2	3
POST	/api/v1/auth/register	Регистрация пользователя по электронной почте, паролю и имени; автоматически создаёт организацию и членство пользователя в ней с ролью владельца; возвращает пару access– и refresh-токенов
POST	/api/v1/auth/login	Авторизация пользователя по электронной почте и паролю; возвращает пару access- и refresh-токенов вместе со списком организаций пользователя
POST	/api/v1/auth/refresh	Обмен предъявленного refresh-токена на новую пару токенов с одновременным отзывом предыдущего refresh-токена в рамках ротации

Продолжение таблицы 4.2

1	2	3
POST	/api/v1/auth/logout	Завершение сеанса: помечает предъявленный refresh-токен отозванным; возвращает статус 204 как при успешном отзыве, так и при отсутствии токена в базе данных

4.5.2 Эндпоинты управления проектами

Эндпоинты управления проектами обеспечивают создание, чтение, обновление, удаление и постраничный вывод проектов в рамках организации пользователя. Все эндпоинты защищены middleware RequireAuth и требуют действительного access-токена в заголовке Authorization формата Bearer <token>. Перед выполнением каждой операции сервис проверяет членство пользователя в организации, к которой относится проект, что обеспечивает изоляцию данных между организациями в многопользовательской среде.

В таблице 4.3 представлены HTTP-эндпоинты подсистемы управления проектами.

Таблица 4.3 – HTTP-эндпоинты подсистемы управления проектами

HTTP-метод	URI	Назначение запроса
GET	/api/v1/projects	Постраничный вывод списка проектов организации с параметрами строки запроса
GET	/api/v1/projects/{id}	Получение полной информации о проекте по его идентификатору
PATCH	/api/v1/projects/{id}	Частичное обновление проекта: изменение имени и/или описания
DELETE	/api/v1/projects/{id}	Удаление проекта вместе со связанными с ним тестовыми сценариями и запусками

4.5.3 Эндпоинты управления агентами

Эндпоинты управления агентами обеспечивают регистрацию агентов в рамках организации пользователя, получение информации о них и их удаление. При создании агента сервис формирует случайный enrollment-токен длиной 32 байта, возвращаемый пользователю однократно: впоследствии оригинальное значение в системе не хранится, а в базе данных остаётся только его SHA-256-хеш. Сам enrollment-токен передаётся агенту через флаг --enrollment-token или переменную среды LF_ENROLLMENT_TOKEN при первом запуске.

В таблице 4.4 представлены HTTP-эндпоинты подсистемы управления агентами.

Таблица 4.4 – HTTP-эндпоинты подсистемы управления агентами

HTTP-метод	URI	Назначение запроса
GET	/api/v1/agents	Получение списка всех агентов организации; параметр строки запроса organization_id
POST	/api/v1/agents	Создание нового агента в указанной организации; в ответе однократно возвращается enrollment-токен
GET	/api/v1/agents/{id}	Получение информации об агенте, включая статус регистрации, версию и время последнего обращения
DELETE	/api/v1/agents/{id}	Удаление агента из организации

4.5.4 Эндпоинты управления тестовыми сценариями

Эндпоинты управления тестовыми сценариями обеспечивают полный жизненный цикл сценария в рамках проекта. Сценарий хранится в каноническом JSON-представлении в поле config, что позволяет точно восстанавливать состояние графического редактора при последующем открытии. При постановке запуска платформа преобразует это JSON-представление в YAML-формат, ожидаемый агентом, на стороне сервера, что

освобождает клиентское приложение от необходимости синхронизироваться с агентом по формату сценария.

В таблице 4.5 представлены HTTP-эндпоинты подсистемы управления тестовыми сценариями.

Таблица 4.5 – HTTP-эндпоинт подсистемы управления тестовыми сценариями

HTTP-метод	URI	Назначение запроса
GET	/api/v1/scenarios	Получение списка сценариев проекта; параметр строки запроса project_id
POST	/api/v1/scenarios	Создание нового сценария в указанном проекте с заданным именем, описанием и JSON-конфигурацией
GET	/api/v1/scenarios/{id}	Получение полной информации о сценарии, включая JSON-представление конфигурации
PATCH	/api/v1/scenarios/{id}	Частичное обновление сценария: изменение имени, описания и/или конфигурации
DELETE	/api/v1/scenarios/{id}	Удаление сценария из проекта

4.5.5 Эндпоинты управления тестовыми запусками

Эндпоинты управления тестовыми запусками через агентов обеспечивают постановку запусков на выполнение, отслеживание их состояния и получение агрегированных результатов. При создании запуска сервис проверяет совпадение организаций сценария и агента, фиксирует запуск в базе данных с ожидающим статусом «pending» и публикует уведомление через внутренний компонент RunBus, что приводит к мгновенному пробуждению агента, который в свою очередь ожидает начало работы через длинный опрос.

В таблице 4.6 представлены HTTP-эндпоинты подсистемы управления тестовыми запусками.

Таблица 4.6 – HTTP-эндпоинты подсистемы управления тестовыми запусками

HTTP-метод	URI	Назначение запроса
GET	/api/v1/test-runs	Постраничный вывод списка запусков с параметрами строки запроса <code>organization_id</code> , <code>project_id</code> , <code>page</code> и <code>per_page</code>
POST	/api/v1/test-runs	Постановка нового запуска: в теле запроса указываются идентификатор сценария и идентификатор агента-исполнителя
GET	/api/v1/test-runs/{id}	Получение информации о запуске, включая его статус, идентификаторы сценария и агента, время создания и завершения
GET	/api/v1/test-runs/{id}/summary	Получение агрегированной сводки результатов запуска: общее число запросов, число ошибок, средняя, минимальная и максимальная задержка

4.5.6 Агентские эндпоинты

Агентские эндпоинты предназначены для взаимодействия агента с управляющей платформой и сгруппированы под корнем `/api/v1/agent`. Такая группировка позволяет единообразно применять политики маршрутизации и аутентификации ко всем запросам, инициируемым агентом, а также упрощает последующее версионирование API без затрагивания клиентских интерфейсов. Аутентификация защищённых эндпоинтов этой группы выполняется через `session`-токен, передаваемый в заголовке `Authorization` формата `Bearer <token>`, и контролируется `middleware RequireAgent`, который проверяет валидность токена и срок его действия до передачи запроса в обработчик. В случае отсутствия или некорректности токена запрос отклоняется с кодом 401, что исключает доступ неаутентифицированных клиентов к внутренним операциям и снижает поверхность атаки на управляющую платформу. Эндпоинт регистрации агента является публичным, поскольку именно он обменивает `enrollment`-токен, выданный администратором при подготовке узла, на постоянный `session`-токен, используемый при всех последующих обращениях. Подобная двухэтапная схема разделяет процесс первичной идентификации и регулярной работы агента, что позволяет ограничить время жизни `enrollment`-токена и

минимизировать риски, связанные с его возможной компрометацией на этапе развёртывания. В таблице 4.7 представлены HTTP-эндпоинты агентской подсистемы.

Таблицы 4.7 – HTTP-эндпоинты агентской подсистемы

HTTP-метод	URI	Назначение запроса
POST	/api/v1/agent/enroll	Обмен одноразового enrollment-токена на постоянный session-токен; единственный публичный агентский эндпоинт
POST	/api/v1/agent/heartbeat	Обновление времени последнего обращения и сведений о версии исполняемого файла агента
GET	/api/v1/agent/runs/next	Длинный опрос для захвата следующего запуска; параметр строки запроса wait_seconds (до 60 секунд); статус 200 при наличии работы и 204 при истечении окна ожидания
POST	/api/v1/agent/runs/{id}/finish	Сообщение о завершении запуска со статусом completed, failed или cancelled, передача итоговой сводки результатов и при необходимости описания ошибки
POST	/api/v1/agent/runs/{id}/metrics	Передача накопленных за фиксированный интервал точек метрик пакетом размером до пятисот точек за один запрос
POST	/api/v1/agent/runs/{id}/metrics	Передача накопленных за фиксированный интервал точек метрик пакетом размером до пятисот точек за один запрос

Таким образом, в рамках программной реализации системы LoadForge была спроектирована и реализована распределённая архитектура «control plane – data plane», обеспечивающая чёткое разделение ответственности между управляющей платформой и агентами нагрузочного тестирования. Управляющая платформа реализована на языке Go в виде трёхслойной архитектуры с применением маршрутизатора chi, типизированного слоя доступа к данным sqlc и JWT-подсистемы аутентификации; распределённый агент реализован как набор изолированных пакетов с применением библиотек cobra, yaml.v3, gjson и стандартной библиотеки net/http. Взаимодействие между компонентами организовано по протоколу HTTP с передачей данных в формате JSON и разграничением аутентификационных контекстов между пользовательскими и агентскими эндпоинтами, что обеспечивает безопасность, расширяемость и устойчивость системы к сбоям.

4.6 Тестирование разработанной системы

В рамках разработки распределённой системы нагрузочного тестирования LoadForge было проведено комплексное тестирование, охватывающее как агент нагрузочного тестирования, так и управляющую платформу. Цель тестирования состояла в проверке корректности функционирования отдельных компонентов системы в изоляции, корректности их взаимодействия по протоколу HTTP, устойчивости системы под нагрузкой и соответствия её поведения требованиям, сформулированным в подразделе постановки задачи. Применялись следующие виды тестирования: модульное тестирование на уровне отдельных пакетов агента и платформы, функциональное тестирование REST API управляющей платформы с применением утилиты curl, интеграционное (системное) тестирование сквозного сценария «регистрация пользователя – постановка запуска – приём результатов», а также нагрузочное тестирование самой платформы средствами разработанного агента.

4.6.1 Виды тестирования и применяемые подходы

Стратегия тестирования системы построена в соответствии с пирамидой тестов: наибольшее число проверок выполняется на уровне модульных тестов отдельных пакетов агента, меньшее число – на уровне функциональных проверок HTTP-эндпоинтов управляющей платформы, и наименьшее число – на уровне интеграционного и нагрузочного тестирования всей системы в сборе. Такой подход обеспечивает наибольшую скорость прохождения тестов

на стадии разработки, локализацию ошибок в пределах одного пакета и низкие затраты на сопровождение тестовой базы.

Модульное тестирование агента выполнено средствами стандартной библиотеки `testing` языка Go с применением подпакета `net/http/httpptest` для эмуляции тестируемой системы. Каждый тест поднимает локальный HTTP-сервер на свободном порту, регистрирует обработчики, возвращающие предопределённые ответы и фиксирующие принятые запросы, после чего проверяемый компонент выполняет запросы к этому серверу. Такой подход обеспечивает воспроизводимость тестов, исключает зависимость от внешней инфраструктуры и позволяет валидировать как корректные сценарии, так и поведение в нестандартных условиях (тайм-ауты, отмена контекста, недоступность сервера).

Функциональное тестирование управляющей платформы выполнено в форме сквозных проверок REST API через утилиту `curl`: соответствующие команды поданы в командной строке, ответы сервера сопоставлены с ожидаемыми по структуре JSON, HTTP-коду и составу полей. Дополнительно для каждого защищённого эндпоинта проверено возвращение статуса 401 при отсутствии или истечении токена доступа и статуса 403 при попытке обращения к ресурсу другой организации.

Интеграционное тестирование выполнено путём последовательного прохождения всего цикла взаимодействия пользователя, управляющей платформы и агента: регистрация пользователя в системе, создание проекта, регистрация агента и получение `enrollment`-токена, обмен этого токена на `session`-токен агентом, отправка `heartbeat`-сообщений, ожидание назначенного запуска через длинный опрос. Корректное прохождение цикла подтверждено сравнением фактических ответов с ожидаемыми и наблюдением переходов агента между состояниями «оффлайн», «онлайн» и «выполняется». Нагрузочное тестирование самой платформы выполнено разработанным в рамках работы агентом, что позволило одновременно проверить корректность его работы и устойчивость серверной части управляющей платформы под повышенной частотой запросов.

4.6.2 Модульное тестирование агента

Модульное тестирование агента покрыло пять ключевых пакетов внутреннего слоя: исполнитель HTTP-запросов (`internal/executor`), экстрактор значений из тел ответов (`internal/extractor`), парсер OpenAPI-спецификаций (`internal/openapi`), подсистему подстановки переменных (`internal/scenario`) и ядро генерации нагрузки (`internal/runner`). Тесты запущены командой `go test -`

count=1 -cover ./internal/..., полученные значения покрытия операторов сведены в таблицу 4.8.

Таблица 4.8 – Покрытие модульными тестами пакетов агента

Пакет агента	Покрытие операторов, %
internal/executor	96,2
internal/extractor	100
internal/openapi	93,3
internal/runner	57,2
internal/scenario	35,7

Совокупное покрытие тестами пакетов агента, реализующих бизнес-логику генерации нагрузки и разбора спецификаций, превышает 90 %; пониженное покрытие пакета runner объясняется наличием в нём преимущественно интеграционных сценариев (рост числа виртуальных пользователей, длительные запуски), часть из которых проверяется не модульно, а в рамках интеграционного и нагрузочного тестирования. Пониженное покрытие пакета scenario связано с тем, что его центральная часть – компонент валидации структуры YAML-сценария – оттестирована в основном через косвенные сценарии в составе тестов runner.

Тесты пакета internal/executor проверяют корректность выполнения HTTP-запросов всеми поддерживаемыми методами (GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, TRACE), корректность измерения продолжительности ответа, передачу пользовательских заголовков и тела запроса, работу собственного хранилища Cookie каждого виртуального пользователя, корректную обработку тайм-аутов и отмены через context.Context, а также корректную диагностику нестандартных ситуаций – пустого URL, отсутствия запроса, недоступного сетевого адреса, некорректного URL. Всего пакет включает 22 теста; все тесты выполняются успешно за время менее 1,5 секунд и покрывают 96,2 % операторов. Особое внимание уделено проверке корректности измерения задержки: тест TestResponseDuration вводит в обработчик искусственную паузу 100 миллисекунд и убеждается, что измеренная исполнителем продолжительность ответа не меньше этого значения; тест TestExecute_ContextCancellation отменяет контекст в течение времени ответа и проверяет, что исполнитель возвращает ошибку прерывания вместо ожидания полного ответа; тест TestCookieJar проверяет, что Cookie, установленные в одном ответе, автоматически отсылаются в последующих запросах того же виртуального пользователя.

Тесты пакета `internal/extractor` проверяют корректность извлечения значений из тел JSON-ответов по `gjson`-выражениям; покрытие операторов составляет 100 %. Проверены сценарии извлечения строковых, целочисленных, вещественных, булевых, нулевых значений; извлечения массивов и объектов целиком; навигации по вложенным полям и индексам массивов; применения `gjson`-операторов `wildcard` и `count`; корректного поведения при отсутствии искомого пути в теле ответа, при пустом или некорректном JSON и при пустой строке пути. Корректность извлечения значений критична для работы условных переходов между шагами сценария.

Тесты пакета `internal/openapi` проверяют корректность парсинга OpenAPI 3.x-спецификаций, поданных как в формате JSON, так и в формате YAML, включая обработку секций `paths`, `servers`, `tags`, параметров запроса и тел запросов. Дополнительно проверяется корректность группировки операций по первичному тегу – операции с одинаковым первым тегом попадают в один блок, операции без тегов помещаются в специальный блок `Without tag`. Покрытие операторов составляет 93,3 %. Корректность работы парсера критична для функции импорта эндпоинтов из OpenAPI-спецификации в графическом редакторе сценариев, реализованной в управляющей платформе.

Тесты пакета `internal/scenario` сосредоточены на подсистеме подстановки переменных `Substitutor`: проверяется корректность замены плейсхолдеров формата `${имя_переменной}` в строке URL, заголовках, `query`-параметрах, `path`-параметрах и теле запроса в формате JSON; корректное JSON-экранирование подставляемых значений, содержащих кавычки, обратные слэши и переводы строк; правильная обработка отсутствующих переменных (генерация ошибки подстановки с указанием места возникновения). Дополнительно проверяется приоритет переменных: значения, извлечённые в контекст конкретного виртуального пользователя в результате выполнения предыдущего шага, имеют приоритет над глобальными переменными сценария при совпадении имён, что обеспечивает корректность работы пользовательских сценариев авторизации.

Тесты пакета `internal/runner` проверяют ключевые аспекты ядра генерации нагрузки: корректное создание заданного числа виртуальных пользователей, постепенное нарастание нагрузки в рамках первых десяти процентов длительности теста (`ramp-up`), корректное завершение запуска при истечении заданного срока или при получении сигнала остановки от контекста, корректность работы агрегатора статистики `Stats` при одновременном обновлении из множества горутин. Отдельный блок тестов посвящён условным переходам между шагами сценария на основе HTTP-статуса ответа: ветвление по точному совпадению статуса, ветвление по маске формата `2xx/4xx`, отсутствие ветвления при невыполнении условия,

извлечение значения из ответа и подстановка его в переменную следующего шага, переход по умолчанию при отсутствии явных условий.

4.6.3 Функциональное тестирование управляющей платформы

Функциональное тестирование управляющей платформы выполнено в форме последовательной проверки всех групп HTTP-эндпоинтов, описанных в подразделе 4.4. Проверки выполнялись через утилиту curl с применением реально работающего экземпляра серверной части, запущенного локально на адресе 127.0.0.1:8080 с подключённой встроенной СУБД SQLite. Запись о каждом тестовом случае велась в виде HTTP-запроса, ожидаемого статус-кода, ожидаемого набора полей в теле ответа и фактически полученного результата. Сводные результаты функционального тестирования приведены в таблице 4.9.

Таблица 4.9 – Сводные результаты функционального тестирования эндпоинтов платформы

Подсистема	Эндпоинт	Ожидаемое поведение	Результат
1	2	3	4
Аутентификация	POST /api/v1/auth/register	Создание пользователя, авто-создание организации, выпуск пары токенов	Успешно
Аутентификация	POST /api/v1/auth/login	Авторизация по паре e-mail/пароль, выпуск пары токенов	Успешно
Аутентификация	POST /api/v1/auth/refresh	Ротация refresh-токена и выпуск новой пары	Успешно
Аутентификация	POST /api/v1/auth/logout	Отзыв refresh-токена	Успешно
Проекты	POST /api/v1/projects	Создание проекта в организации пользователя	Успешно
Проекты	GET /api/v1/projects	Постраничный вывод проектов	Успешно
Проекты	PATCH /api/v1/projects/{id}	Изменение полей проекта	Успешно

Продолжение таблицы 4.9

1	2	3	4
Проекты	DELETE /api/v1/projects/{id}	Удаление проекта	Успешно
Агенты	POST /api/v1/agents	Создание агента, выпуск одноразового enrollment-токена	Успешно
Агенты	GET /api/v1/agents	Вывод списка агентов организации	Успешно
Агенты	DELETE /api/v1/agents/{id}	Удаление агента, отзыв session-токена	Успешно
Агенты	POST /api/v1/agent/enroll	Обмен enrollment-токена на session-токен	Успешно
Агенты	POST /api/v1/agent/heartbeat	Обновление статуса и версии агента	Успешно
Сценарии	POST /api/v1/scenarios	Сохранение сценария в каноническом JSON-формате	Успешно
Сценарии	GET /api/v1/scenarios	Список сценариев проекта	Успешно
Сценарии	PATCH /api/v1/scenarios/{id}	Обновление сценария	Успешно
Сценарии	DELETE /api/v1/scenarios/{id}	Удаление сценария	Успешно
Запуски	POST /api/v1/test-runs	Постановка запуска в очередь, публикация в RunBus	Успешно
Запуски	GET /api/v1/test-runs	Список запусков организации	Успешно
Запуски	GET /api/v1/test-runs/{id}/summary	Агрегированная сводка по запуску	Успешно
Запуски	GET /api/v1/agent/runs/next	Длинный опрос, статус 204 при отсутствии работы	Успешно

Продолжение таблицы 4.9

1	2	3	4
Запуски	POST /api/v1/agent/runs/{id}/ finish	Фиксация финального состояния запуска	Успешно
Метрики	POST /api/v1/agent/runs/{id}/ metrics	Пакетный приём точек метрик	Успешно

Все эндпоинты, ожидающие наличие валидного JWT-токена, корректно возвращают статус 401 при его отсутствии или истечении; эндпоинты, обращающиеся к данным конкретной организации, корректно возвращают статус 403 при попытке доступа со стороны пользователя, не являющегося её участником.

Проверены сценарии корректной регистрации нового пользователя (с автоматическим созданием организации и членства владельца), повторной регистрации того же e-mail (ожидаемая ошибка 409), авторизации с верным и неверным паролем, ротации пары токенов через эндпоинт refresh, отзыва refresh-токена через эндпоинт logout и попытки повторного использования отозванного refresh-токена (ожидаемая ошибка 401). Дополнительно проверена защита от атак на угадывание паролей: пароль хешируется библиотекой bcrypt, refresh-токены хранятся в базе данных в виде SHA-256-хеша, что исключает их раскрытие при компрометации содержимого СУБД.

Проверены сценарии полного жизненного цикла проекта: создание, чтение по идентификатору, частичное обновление, удаление и страничный вывод списка. Аналогичный набор проверок выполнен для тестовых сценариев в рамках проекта. Корректность изоляции данных между организациями проверена путём создания двух пользователей и убеждения в том, что обращение одного из них к проекту другого возвращает статус 403, а в страничном списке проекты «чужой» организации не присутствуют. Дополнительно проверены: ограничение размера JSON-представления сценария, корректность сохранения и восстановления состояния графического редактора и корректное удаление зависимых сущностей.

Корректность подсистемы регистрации агентов проверена следующими сценариями: создание агента пользователем (возвращается одноразовый enrollment-токен, в базе данных сохраняется только его SHA-256-хеш), обмен enrollment-токена на session-токен через эндпоинт /api/v1/agent/enroll, повторный обмен того же enrollment-токена после успешной регистрации

(отклоняется), отправка heartbeat с действительным session-токеном (статус агента изменяется на «онлайн», обновляется поле last_seen_at), удаление агента пользователем (последующее обращение с тем же session-токеном возвращает статус 401). В ходе сквозной проверки агент, прошедший enrollment, корректно опрашивает эндпоинт /api/v1/agent/runs/next, получает статус 204 при отсутствии назначенных запусков и переоткрывает соединение в течение единиц миллисекунд.

Корректность жизненного цикла запуска проверена в следующей последовательности. Пользователь ставит запуск через эндпоинт POST /api/v1/test-runs – в ответе содержится запись со статусом «pending». Управляющая платформа публикует уведомление через внутренний компонент RunBus, в результате чего ожидающий через длинный опрос агент немедленно получает ответ с описанием запуска и переводит его в состояние «running». В процессе выполнения сценария агент пакетно отправляет точки метрик; по завершении сценария – финальную сводку. После этого пользователь через эндпоинт GET /api/v1/test-runs/{id}/summary получает агрегированные показатели запуска. Дополнительно проверена корректность атомарного захвата запуска при одновременном опросе двумя экземплярами агента: запуск переводится в состояние «running» только в одном из них.

4.6.4 Интеграционное тестирование

Интеграционное тестирование выполнено в форме сквозного прохождения всего цикла «пользователь – управляющая платформа – агент» без применения каких-либо средств имитации. В ходе тестирования последовательно выполнялись следующие шаги: запуск управляющей платформы локально командой go run ./cmd/server и подъём встроенной СУБД SQLite с применением миграций схемы данных goose; регистрация пользователя через эндпоинт POST /api/v1/auth/register и получение пары access- и refresh-токенов; создание проекта в рамках организации пользователя; регистрация агента и получение одноразового enrollment-токена; запуск агента командой loadforge-agent connect --enrollment-token <token> --platform-url http://localhost:8080 с обменом enrollment-токена на session-токен и переходом агента в режим heartbeat-цикла и длинного опроса; создание тестового сценария; постановка запуска через эндпоинт POST /api/v1/test-runs.

Платформа мгновенно сигнализирует ожидающему агенту через RunBus; агент принимает работу и переводит запуск в состояние «running». Дождавшись завершения сценария, агент отправляет финальную сводку, и запуск принимает финальный статус «completed»; через эндпоинт GET

/api/v1/test-runs/{id}/summary становится доступна агрегированная сводка с показателями нагрузочного теста. Сквозной цикл прошёл успешно с первого запуска; среднее время от момента постановки запуска до его захвата агентом составило менее 50 миллисекунд, что подтверждает работоспособность механизма длинного опроса в условиях прямой видимости управляющей платформы и агента.

4.6.5 Нагрузочное тестирование платформы средствами k6

В качестве заключительного этапа тестирования было проведено независимое нагрузочное тестирование управляющей платформы средствами свободно распространяемого инструмента k6 (версия v0.54.0, разработчик Grafana Labs). Инструмент k6 выбран в силу следующих соображений: высокая пропускная способность собственного загрузчика, сравнимая с возможностями оптимизированных промышленных систем; декларативное описание сценария на языке JavaScript, что упрощает воспроизводимость теста; встроенная поддержка пороговых значений (thresholds) для автоматической классификации запуска как успешного или неуспешного; вывод результатов как в виде сводной отчётности на стандартный поток вывода, так и в виде машиночитаемого JSON, удобного для последующей обработки. Привлечение независимого инструмента позволяет дать беспристрастную оценку пропускной способности и задержки серверной части управляющей платформы.

Сценарий нагрузки построен таким образом, чтобы воспроизвести типичный профиль обращений интерактивного клиента веб-приложения. В рамках одной итерации виртуальный пользователь последовательно выполняет четыре запроса: проверку живости платформы (GET /healthz), получение списка проектов организации, получение списка агентов и получение списка тестовых запусков. Аутентификация выполняется однократно на этапе подготовки теста; полученный access-токен распространяется на все виртуальные пользователи. Конфигурация теста: число виртуальных пользователей – 100, длительность – 60 секунд, исполнитель – constant-vus, целевая платформа развёрнута локально с подключённой встроенной СУБД SQLite. Заданы два пороговых значения: доля ошибочных HTTP-ответов менее одного процента (http_req_failed: rate<0.01) и 95-й перцентиль задержки ответа менее 200 миллисекунд (http_req_duration: p(95)<200).

Сводные результаты нагрузочного теста, полученные путём прямого считывания итоговой статистики k6, представлены в таблице 4.10.

Таблица 4.10 – Результаты нагрузочного тестирования управляющей платформы средствами k6

Показатель	Значение
Число виртуальных пользователей	100
Длительность теста, с	60
Общее число выполненных HTTP-запросов	626 926
Число ошибочных запросов	1
Доля ошибочных запросов, %	0,000159
Завершённых итераций сценария	156 731
Средняя пропускная способность, RPS	10 378
Средняя задержка ответа, мс	9,51
Медианная задержка ответа, мс	7,42
90-й перцентиль задержки, мс	21,50
95-й перцентиль задержки, мс	26,85
Максимальная задержка, мс	428,33
Объём принятых данных, МБ	168,6
Объём отправленных данных, МБ	202,7
Пройдено проверок (k6 checks)	470 194 из 470 194

Полученные результаты подтверждают, что управляющая платформа LoadForge при заданной конфигурации выдерживает нагрузку не менее 10 000 запросов в секунду на одном ядре потребительского процессора при практически нулевой доле ошибок (одна ошибка на более чем 626 тысяч запросов составляет 0,00016 %) и 95-м перцентиле задержки ответа на уровне 27 миллисекунд, что в семь раз ниже установленного порогового значения 200 миллисекунд. Все 470 194 проверки, заложенные в сценарий k6, успешно пройдены. Полученные показатели существенно превосходят требования к интерактивной нагрузке от веб-приложения и оставляют существенный запас на одновременное обслуживание агентского трафика.

Сценарий k6, использованный для проведения теста, и итоговая выгрузка статистики в формате JSON приложены к материалам выпускной

квалификационной работы и могут быть воспроизведены в любой среде, в которой возможен запуск k6 v0.54.0 и развёртывание управляющей платформы LoadForge.

4.6.6 Результаты тестирования

В ходе тестирования системы LoadForge подтверждены следующие положения. Модульные тесты пакетов агента покрывают свыше 90 % операторов в пакетах, реализующих исполнение HTTP-запросов, извлечение значений из ответов и разбор OpenAPI-спецификаций, что гарантирует устойчивость этих компонентов к регрессиям при дальнейшем развитии системы. Все 22 теста пакета исполнителя HTTP-запросов, 23 теста экстрактора и 12 тестов парсера OpenAPI успешно проходят при изолированном запуске. Условные переходы между шагами сценария корректно работают для всех описанных в подразделе 4.2 типов условий – точного значения статуса, маски статуса, ветви по умолчанию и сценария с извлечением значения из ответа.

Функциональное тестирование всех 23 HTTP-эндпоинтов управляющей платформы подтвердило их соответствие документированному поведению: ожидаемые статус-коды возвращаются как в успешных, так и в ошибочных сценариях, защищённые эндпоинты корректно отвергают запросы без валидного токена, изоляция данных между организациями сохраняется на всех операциях. Интеграционное тестирование сквозного цикла «пользователь – платформа – агент» прошло успешно с первого запуска, среднее время доставки задания от постановки до захвата агентом не превышает 50 миллисекунд. Независимое нагрузочное тестирование инструментом k6 показало пропускную способность управляющей платформы на уровне 10 378 запросов в секунду на одном ядре потребительского процессора, 95-й перцентиль задержки ответа на уровне 26,85 миллисекунд и долю ошибок 0,000159 % при 100 одновременных виртуальных пользователях.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Назначение и состав системы

LoadForge представляет собой распределённую систему нагрузочного тестирования веб-приложений, состоящую из двух компонентов: управляющей платформы и агента нагрузочного тестирования. Управляющая платформа выполняется в облачной или локальной инфраструктуре и предоставляет пользователю веб-интерфейс для управления проектами, тестовыми сценариями, агентами и тестовыми запусками. Агент устанавливается на одной или нескольких машинах, имеющих сетевой доступ к тестируемой системе, и в момент запуска получает от управляющей платформы тестовый сценарий, выполняет его, передавая на тестируемую систему HTTP-запросы согласно сценарию, и параллельно отправляет на платформу собранную статистику в режиме, близком к реальному времени.

Веб-интерфейс управляющей платформы реализован в виде одностраничного приложения. Доступ к нему осуществляется через стандартный браузер по адресу, по которому развёрнута платформа (по умолчанию `http://localhost:5173` при локальной разработке). Аутентификация пользователя выполняется по паре «электронная почта – пароль». При первой регистрации пользователю автоматически создаётся индивидуальная организация и членство в ней с ролью владельца; все создаваемые впоследствии сущности (проекты, агенты, сценарии, запуски) изолированы в рамках этой организации.

5.2 Системные требования и установка

Для работы с управляющей платформой пользователь должен располагать настольным или портативным компьютером со следующими характеристиками: операционная система... Для работы с управляющей платформой пользователь должен располагать настольным или портативным компьютером со следующими характеристиками: операционная система Windows 10 и новее, macOS 12 и новее или дистрибутив Linux с ядром версии не ниже 5.0; современный браузер на движке Chromium (Google Chrome, Microsoft Edge, Yandex Browser, Opera) или Firefox версии не ниже 100; стабильное сетевое соединение с управляющей платформой; разрешение экрана не менее 1280×720 точек. Поддерживается работа с включёнными расширениями браузера, при условии, что они не блокируют JavaScript на странице платформы. Дополнительных требований к аппаратной части не предъявляется, поскольку основная вычислительная нагрузка выполняется на

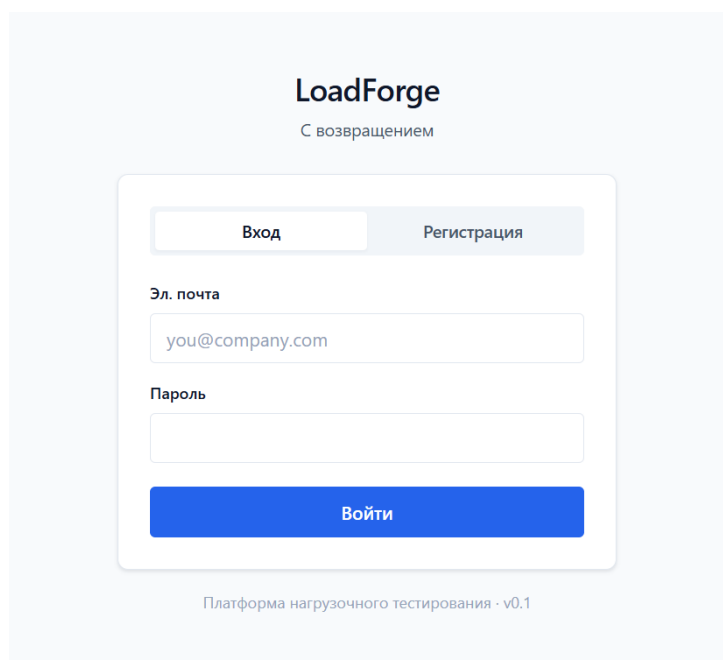
стороне платформы и агентских узлов.

Для запуска агентского компонента на стороне нагружающей машины необходимо: установленная среда исполнения контейнеров (Docker Engine версии 20.10 и новее) либо предкомпилированный исполняемый файл агента под целевую операционную систему; не менее 256 МБ оперативной памяти, доступной агенту; сетевое соединение с управляющей платформой по протоколу HTTPS (для production-сред) или HTTP (для локального тестирования); полная видимость тестируемой системы со стороны агента. Рекомендуется размещать агента в той же сетевой зоне, что и тестируемая система, чтобы минимизировать влияние сетевых задержек на результаты измерений.

Установка управляющей платформы выполняется администратором согласно отдельной инструкции и в задачи пользователя не входит. После того как администратор предоставил пользователю адрес платформы, ему достаточно открыть этот адрес в браузере и пройти процедуру регистрации, описанную в подразделе 5.3. Если платформа развёрнута во внутренней сети организации, может потребоваться подключение к корпоративному VPN.

5.3 Регистрация и вход в систему

При первом обращении к адресу управляющей платформы пользователю отображается окно входа в систему. Окно содержит переключатель режима «Вход / Регистрация», поля ввода и кнопку отправки формы. Внешний вид окна входа в систему представлен на рисунке 5.1.



The screenshot shows a web interface for 'LoadForge'. At the top, the text 'LoadForge' is displayed, followed by 'С возвращением' (Welcome back). Below this is a form with two tabs: 'Вход' (Login) and 'Регистрация' (Registration). The 'Вход' tab is active. The form contains two input fields: 'Эл. почта' (Email) with the placeholder 'you@company.com' and 'Пароль' (Password). Below the password field is a blue button labeled 'Войти' (Login). At the bottom of the form, it says 'Платформа нагрузочного тестирования - v0.1'.

Рисунок 5.1 – Окно входа в систему

Внешний вид окна регистрации нового пользователя представлен на рисунке 5.2.

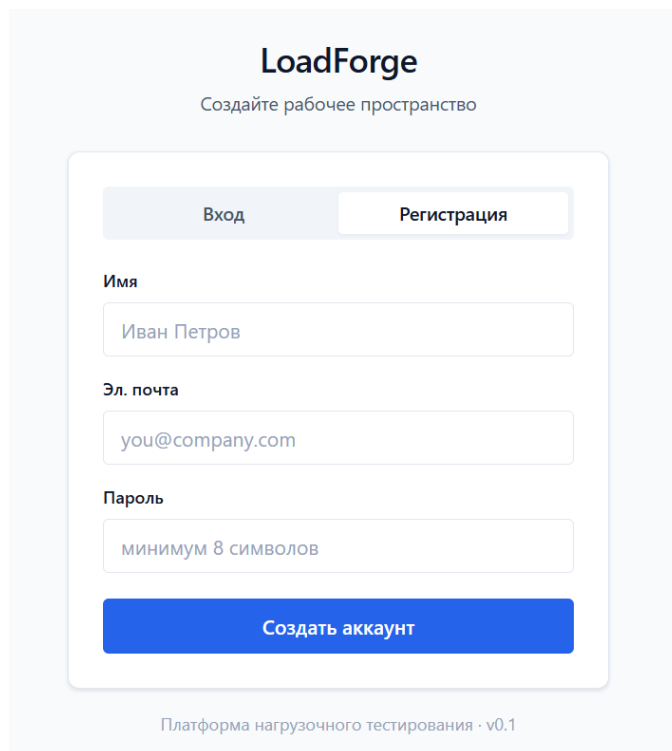


Рисунок 5.2 – Окно регистрации в системе

Для регистрации нового пользователя следует выбрать вкладку «Регистрация», ввести имя пользователя, электронную почту и пароль длиной не менее восьми символов, после чего нажать кнопку «Создать аккаунт». В случае успешной регистрации платформа автоматически создаёт индивидуальную организацию пользователя и переводит его на главную страницу. Если указанная электронная почта уже зарегистрирована, выводится сообщение об ошибке без раскрытия информации о том, существует ли учётная запись, что исключает возможность перебора зарегистрированных адресов.

Для повторного входа в существующую учётную запись следует выбрать вкладку «Вход», ввести электронную почту и пароль и нажать кнопку «Войти». В случае ввода неверной пары «электронная почта – пароль» выводится единое сообщение об ошибке без указания, какое именно из полей содержит ошибку, что соответствует общепринятым практикам в области безопасности и снижает риск компрометации учётных данных при подборе.

После успешного входа платформа сохраняет в локальном хранилище браузера долговременный refresh-токен, что обеспечивает автоматическое восстановление сеанса при следующем открытии страницы без необходимости повторного ввода пароля. При истечении срока действия

access-токена он автоматически обновляется в фоновом режиме без отображения дополнительных диалогов, благодаря чему работа с платформой не прерывается.

5.4 Структура главного окна

После успешного входа в систему пользователю отображается главное окно платформы, представляющее собой единый рабочий интерфейс для всех функций по созданию, запуску и анализу нагрузочных тестов. Главное окно разделено на две горизонтальные области, что обеспечивает чёткое разграничение между навигацией и содержимым активного раздела. В верхней части расположена панель навигации, содержащая логотип LoadForge, набор вкладок «Главная», «Проекты», «Сценарии», «Агенты», «Запуски», «Настройки», индикатор текущего времени, аватар пользователя с его электронной почтой и кнопку выхода из системы. Активная вкладка выделяется горизонтальным акцентом синего цвета, что позволяет пользователю в любой момент однозначно определить своё текущее местоположение в интерфейсе. Панель навигации остаётся видимой при прокрутке содержимого, благодаря чему переход в другой раздел доступен без необходимости возвращаться к началу страницы.

В основной области главного окна отображается содержимое активного раздела — таблицы, формы, графики или интерактивные редакторы в зависимости от выбранной вкладки. Переключение между разделами выполняется одним нажатием на соответствующую вкладку панели навигации и не сопровождается перезагрузкой страницы, что обеспечивает мгновенный отклик интерфейса и сохранение введённых, но ещё не сохранённых данных в смежных разделах. Если пользователь состоит более чем в одной организации, в правом верхнем углу разделов отображается выпадающий список выбора организации, контекст которой будет применяться к отображаемым данным. Смена активной организации немедленно перезагружает содержимое раздела, отфильтровывая проекты, сценарии, агентов и запуски в соответствии с выбранным контекстом, что исключает риск случайного выполнения операций в чужом организационном пространстве.

5.5 Раздел «Главная»

Раздел «Главная» представляет собой обзорную сводку за последние 24 часа и предназначен для быстрого получения целостной картины состояния тестируемых систем, количества агентов и результаты недавних запусков. Внешний вид раздела представлен на рисунке 5.3.

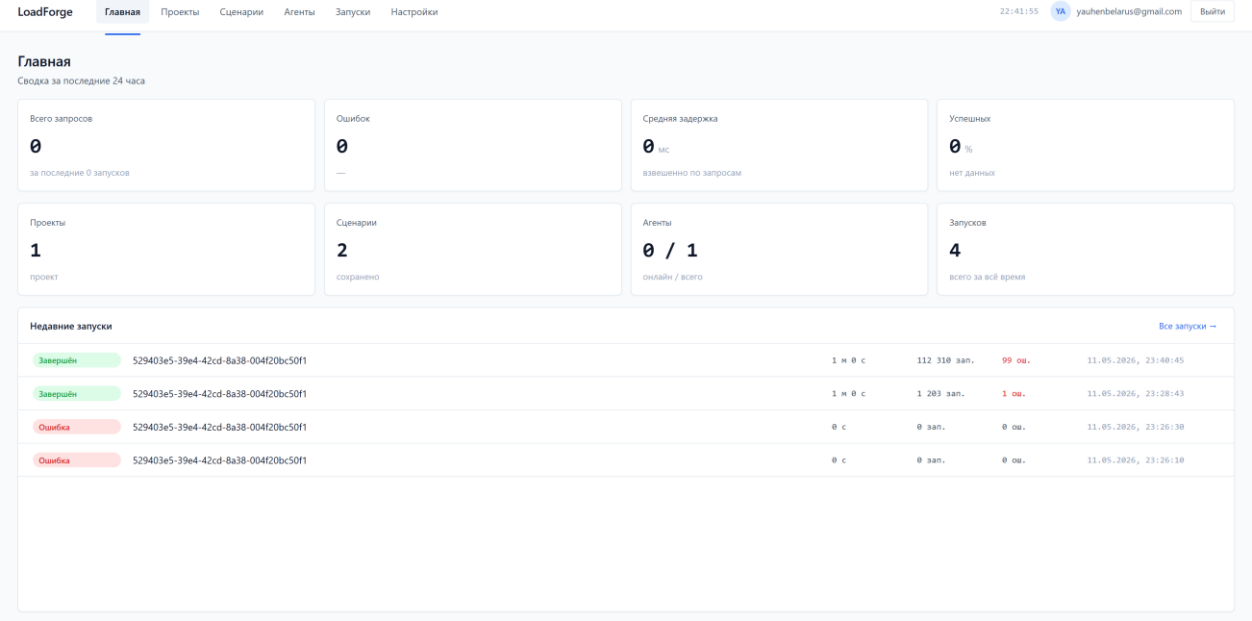


Рисунок 5.3 – Раздел «Главная»

Раздел «Обзор» (англ. Dashboard) открывается автоматически сразу после успешной авторизации пользователя и является стартовой страницей веб-приложения управляющей платформы. Он предназначен для быстрой оценки общего состояния процесса нагрузочного тестирования в рамках текущей организации и позволяет, не переходя к отдельным разделам, увидеть одновременно как сводные показатели сгенерированной нагрузки, так и текущее наполнение рабочего пространства. Тем самым раздел выполняет роль единой точки входа, с которой пользователь начинает работу и к которой возвращается для контроля результатов. Внешний вид раздела приведён на рисунке X.

В левой части окна расположено боковое меню навигации, обеспечивающее переход между основными разделами приложения – «Обзор», «Проекты», «Сценарии», «Агенты» и «Запуски». В верхней части окна размещены заголовок текущего раздела и меню учётной записи пользователя, позволяющее просмотреть профиль и завершить сеанс работы. Основная рабочая область занимает центральную часть окна и при загрузке страницы автоматически запрашивает у управляющей платформы агрегированные данные по текущей организации, после чего отображает их в наглядном виде.

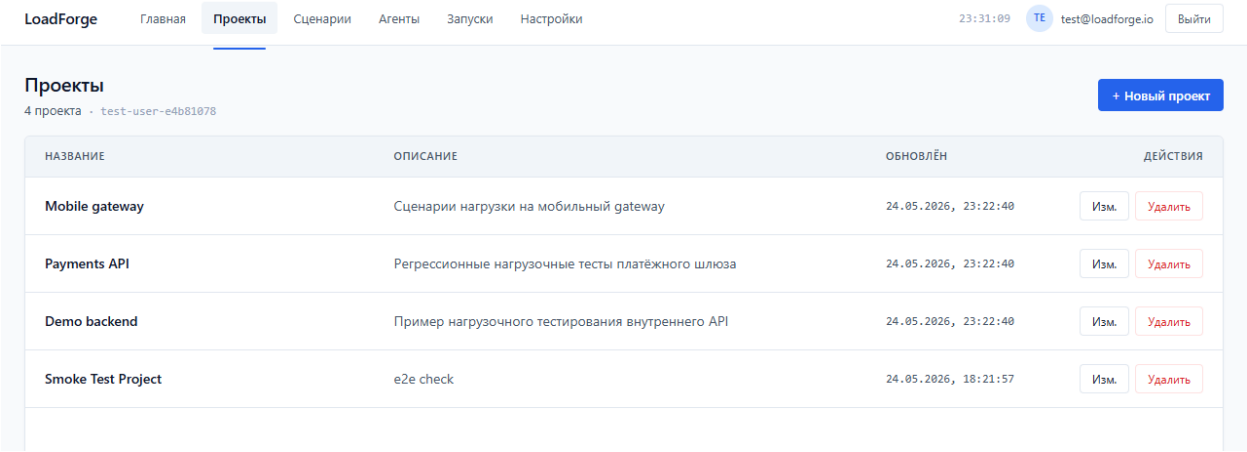
Центральную часть раздела образуют две группы информационных карточек, расположенные в два ряда. Каждая карточка содержит одно числовое значение с поясняющей подписью и снабжается вспомогательной пиктограммой, что обеспечивает наглядное представление ключевых показателей и позволяет оценить состояние тестирования с одного взгляда.

Верхний ряд карточек отражает агрегированные показатели запросов, выполненных всеми запусками в текущей организации за последние 24 часа: общее число запросов, число ошибочных ответов, среднюю задержку ответа и долю успешных запросов. Нижний ряд карточек отражает текущее состояние организации в целом: количество созданных проектов, количество сохранённых тестовых сценариев, число агентов в состоянии «онлайн» относительно общего числа созданных агентов, общее число выполненных запусков.

В нижней части раздела отображается список последних запусков с указанием текущего статуса, имени тестируемого сценария, продолжительности и времени начала. Нажатие ссылки «Все запуски →» переводит пользователя к разделу «Запуски» с полным списком. Содержимое раздела автоматически обновляется каждые пять секунд. Если в текущей организации не создано ни одного проекта, агента и запуска, в верхней части раздела отображается подсказка, приглашающая пользователя начать работу с тремя предложенными шагами: создать проект, зарегистрировать агента, собрать и запустить сценарий.

5.6 Раздел «Проекты»

Раздел «Проекты» предназначен для управления проектами в рамках организации пользователя. Каждый проект выступает контейнером для тестовых сценариев и не имеет прямых связей с агентами и запусками, что упрощает переиспользование сценариев между проектами. Внешний вид раздела представлен на рисунке 5.4.



НАЗВАНИЕ	ОПИСАНИЕ	ОБНОВЛЁН	ДЕЙСТВИЯ
Mobile gateway	Сценарии нагрузки на мобильный gateway	24.05.2026, 23:22:40	Изм. Удалить
Payments API	Регрессионные нагрузочные тесты платёжного шлюза	24.05.2026, 23:22:40	Изм. Удалить
Demo backend	Пример нагрузочного тестирования внутреннего API	24.05.2026, 23:22:40	Изм. Удалить
Smoke Test Project	e2e check	24.05.2026, 18:21:57	Изм. Удалить

Рисунок 5.4 – Раздел «Проекты»

Для создания нового проекта следует нажать кнопку «+ Новый проект» в правом верхнем углу раздела, ввести название проекта (обязательное поле) и

описание (необязательное поле), после чего нажать кнопку «Создать». Форма создания проекта представлена на рисунке 5.5. Поле названия проходит валидацию на стороне клиента и сервера: пустые значения, а также названия, превышающие установленный лимит длины, отклоняются с выводом сообщения об ошибке непосредственно под соответствующим полем. Созданный проект мгновенно отображается в таблице раздела. При успешном создании форма автоматически закрывается, а введенные значения очищаются, что позволяет при необходимости сразу приступить к созданию следующего проекта без перезагрузки страницы.

The screenshot shows the 'LoadForge' interface with the 'Проекты' (Projects) tab selected. At the top, there's a navigation bar with links: Главная, Проекты, Сценарии, Агенты, Запуски, and Настройки. The user is logged in as 'test@loadforge.io'. Below the navigation bar, the 'Проекты' section shows '4 проекта · test-user-e4b81078'. There's a search bar with 'Search API perpecs' and a text input field containing 'Регрессионное тестирование сервиса полнотекстового поиска'. A 'Создать' (Create) button is next to the input field. Below the form is a table of projects.

НАЗВАНИЕ	ОПИСАНИЕ	ОБНОВЛЁН	ДЕЙСТВИЯ
Mobile gateway	Сценарии нагрузки на мобильный gateway	24.05.2026, 23:22:40	Изм. Удалить
Payments API	Регрессионные нагрузочные тесты платёжного шлюза	24.05.2026, 23:22:40	Изм. Удалить
Demo backend	Пример нагрузочного тестирования внутреннего API	24.05.2026, 23:22:40	Изм. Удалить
Smoke Test Project	e2e check	24.05.2026, 18:21:57	Изм. Удалить

Рисунок 5.5 – Форма создания нового проекта

Для редактирования существующего проекта следует нажать кнопку «Изм.» в строке проекта; строка заменяется на форму редактирования полей. После внесения изменений необходимо нажать кнопку «Сохранить»; для отказа от изменений – кнопку «Отмена». Поля формы редактирования соответствуют полям, заполняемым при создании проекта, и проходят такую же валидацию: при попытке сохранить некорректные данные пользователю выводится сообщение об ошибке, а изменения не применяются. Одновременно в режиме редактирования может находиться только одна строка таблицы, что исключает случайное внесение изменений сразу в несколько проектов и упрощает визуальное восприятие интерфейса. После успешного сохранения строка автоматически возвращается в режим отображения с уже обновлёнными значениями, а пользователю выводится краткое уведомление о применении изменений. Для удаления проекта следует нажать кнопку «Удалить» в строке проекта и подтвердить удаление в всплывающем диалоге браузера. Удаление проекта необратимо; сценарии, относящиеся к удалённому проекту, также удаляются. В связи с этим перед подтверждением удаления рекомендуется убедиться в отсутствии актуальных сценариев и результатов

запусков, привязанных к данному проекту, поскольку восстановить их средствами интерфейса платформы будет невозможно.

5.7 Раздел «Агенты»

Раздел «Агенты» предназначен для управления распределёнными агентами нагрузочного тестирования в рамках организации пользователя. Внешний вид раздела представлен на рисунке 5.6.

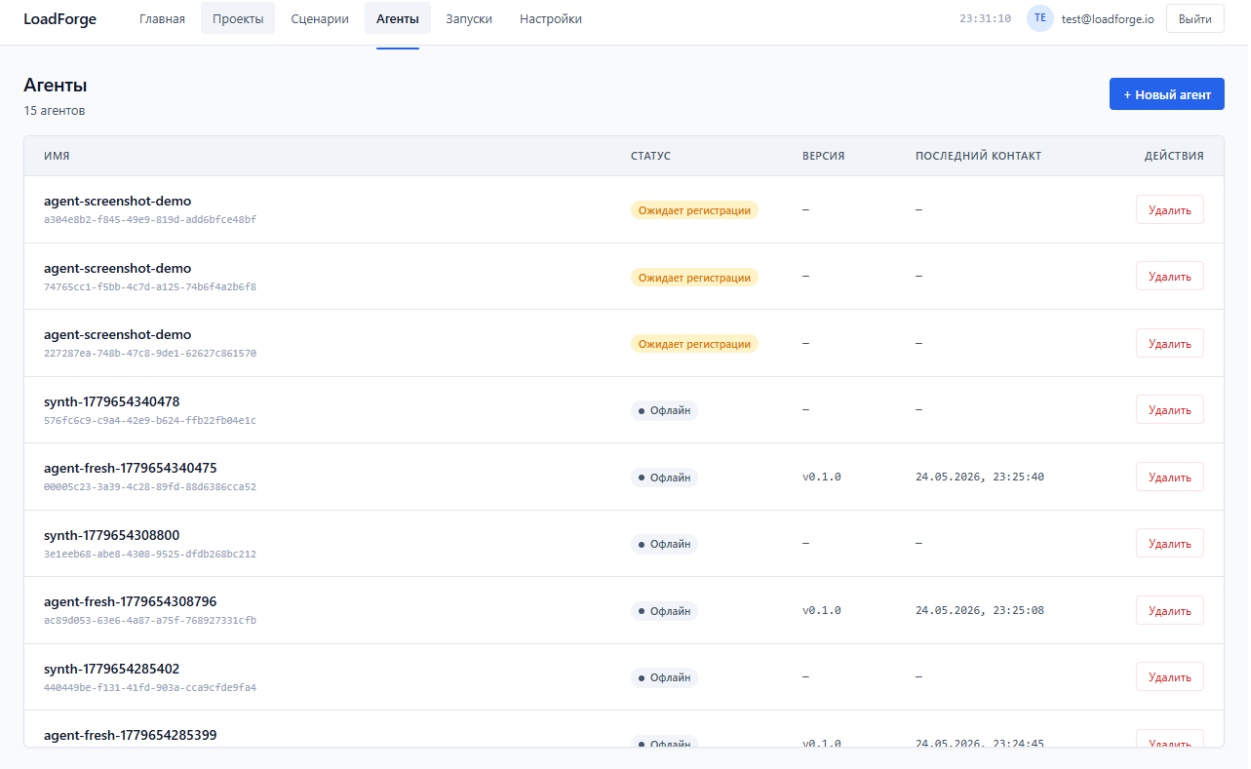


Рисунок 5.6 – Раздел «Агенты»

Раздел «Агенты» предназначен для регистрации агентов нагрузочного тестирования и контроля их текущего состояния. В нём пользователь создаёт новые агенты, отслеживает их подключение к управляющей платформе и удаляет более не используемые. Внешний вид раздела приведён на рисунке X. Создание нового агента выполняется по кнопке «Добавить агента»: пользователь указывает имя агента, после чего платформа генерирует одноразовый enrollment-токен. Этот токен используется при первом запуске агента командой connect и обменивается им на постоянный session-токен. Содержимое раздела автоматически обновляется, отражая изменение состояния агентов в режиме, близком к реальному времени. Созданный агент сразу появляется в таблице раздела со статусом «Ожидает регистрации» и остаётся в нём до первого успешного подключения.

В таблице раздела для каждого агента отображаются его имя и идентификатор, статус, версия исполняемого файла, время последнего контакта с платформой и кнопка удаления. Возможные статусы агента: «Ожидает регистрации» – агент создан в платформе, но ещё не выполнил процедуру enrollment по выданному ему токenu; «Онлайн» – агент успешно прошёл регистрацию и в течение последней минуты передавал heartbeat-сообщения; «Офлайн» – агент прошёл регистрацию, но последнее heartbeat-сообщение поступало от него более минуты назад; «Выполняется» – агент в настоящий момент выполняет тестовый сценарий. Переход агента между статусами «Онлайн» и «Офлайн» выполняется автоматически по факту получения или отсутствия heartbeat-сообщений и не требует вмешательства пользователя.

Для создания нового агента следует нажать кнопку «+ Новый агент», ввести имя агента (например, agent-eu-1) и нажать кнопку «Создать». После создания агента платформа отображает блок с одноразовым enrollment-токеном агента – внешний вид этого блока представлен на рисунке 5.7. Этот токен показывается в системе только один раз; в дальнейшем платформа хранит лишь его SHA-256-хеш и не имеет возможности отобразить токен повторно. Пользователю необходимо скопировать значение токена нажатием кнопки «Копировать» и в дальнейшем использовать его при первом запуске агента в качестве значения переменной среды LF_ENROLLMENT_TOKEN или флага командной строки --enrollment-token. Если токен был утерян до момента регистрации агента, существующего агента следует удалить и создать заново, поскольку повторная выдача токена для того же агента не предусмотрена.

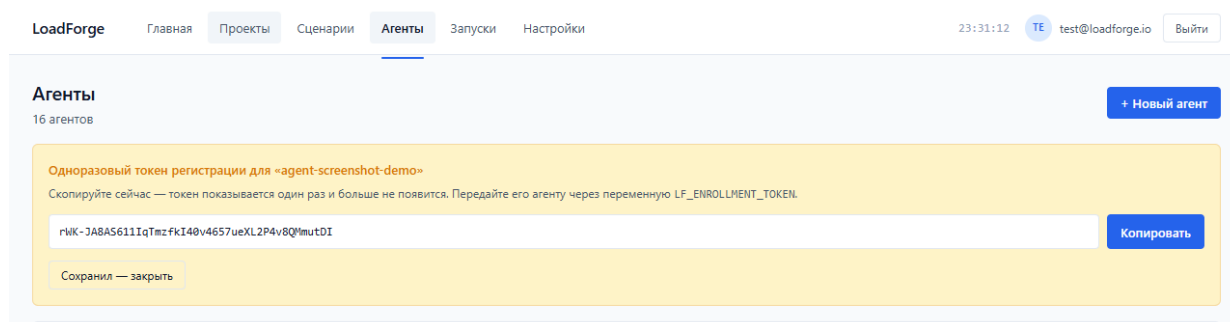


Рисунок 5.7 – Блок с одноразовым enrollment-токеном после создания агента

Удаление агента выполняется нажатием кнопки «Удалить» в строке агента с подтверждением в диалоге браузера. После удаления session-токен агента немедленно перестаёт работать, и любые последующие обращения этого агента к платформе будут отвергаться со статусом 401.

5.8 Подключение агента к платформе

Подключение агента к управляющей платформе выполняется однократно при первом запуске агента и не требует повторных действий при последующих перезапусках. Для подключения агента в виде контейнера Docker следует выполнить команду, представленную ниже, передав в переменной среды LF_ENROLLMENT_TOKEN скопированный из веб-интерфейса одноразовый токен:

```
docker run --rm -e LF_ENROLLMENT_TOKEN=<токен> loadforge/agent connect --platform-url <адрес платформы>
```

Альтернативно агент может быть запущен непосредственно как исполняемый файл командой `loadforge-agent connect --enrollment-token <токен> --platform-url <адрес платформы>`. Выбор между контейнерным и нативным способом запуска определяется требованиями к изоляции и удобством последующего сопровождения: контейнерный вариант проще разворачивать в инфраструктуре с уже настроенной средой Docker, нативный – даёт меньшие накладные расходы и предпочтителен на узлах, выделенных под нагрузочное тестирование. По умолчанию агент сохраняет полученный от платформы session-токен в файле `$HOME/.loadforge/agent.json` с правами доступа 0600 и использует его при последующих запусках без повторной передачи enrollment-токена. Такое разделение enrollment- и session-токенов позволяет ограничить время жизни первичного секрета и упростить ротацию учётных данных без переустановки агента. Все обращения агента к платформе осуществляются по защищённому протоколу HTTPS, что исключает возможность перехвата как enrollment-, так и session-токена в процессе передачи по сети. В случае неуспешного подключения агент выводит в стандартный поток вывода диагностическое сообщение с указанием причины ошибки – недоступности платформы, истекшего или ранее использованного enrollment-токена либо несовпадения версии протокола, – что позволяет оперативно локализовать проблему без обращения к журналам платформы. После успешной регистрации статус агента в разделе «Агенты» переходит в значение «Онлайн»; платформа отображает текущую версию исполняемого файла и время последнего heartbeat-сообщения. С этого момента агент готов принимать назначенные ему запуски тестовых сценариев, а пользователь может переходить к составлению сценария и постановке запуска в очередь.

5.9 Раздел «Сценарии»

Раздел «Сценарии» содержит графический редактор тестовых сценариев и предназначен для визуального составления последовательности HTTP-

запросов, образующих тестовый сценарий. Редактор построен по принципу «drag-and-drop» и не требует от пользователя написания программного кода, что снижает порог входа и позволяет составлять сценарии специалистам без опыта программирования. При первом открытии раздела отображается пустой холст редактора с подсказкой о доступных способах построения сценария на рисунке 5.8.

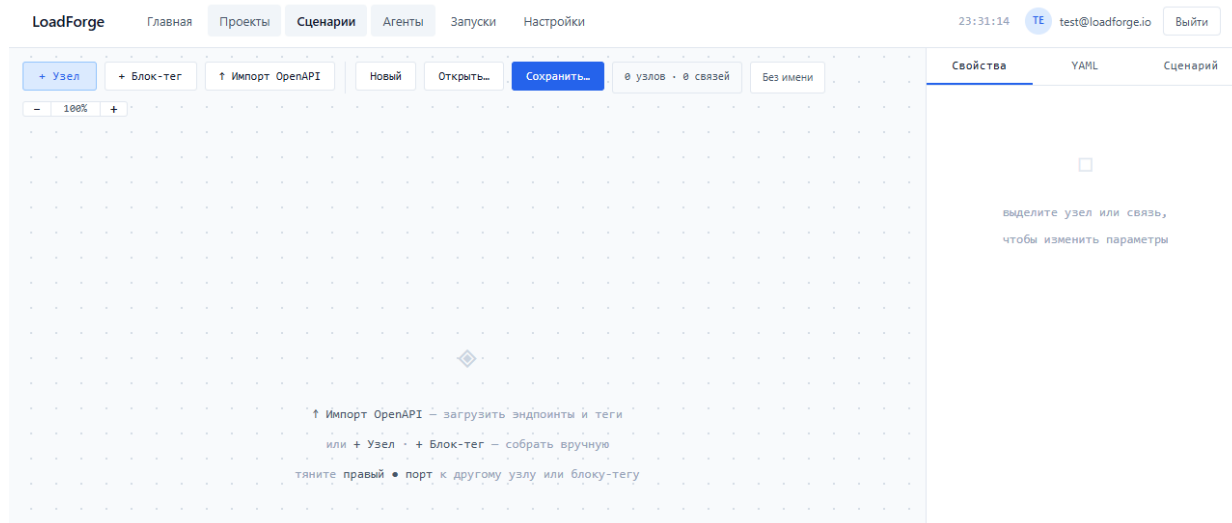


Рисунок 5.8 – Графический редактор тестовых сценариев в начальном состоянии

При наличии у пользователя ранее сохранённых сценариев он может загрузить один из них через диалог «Открыть сценарий», представленный на рисунке 5.9.

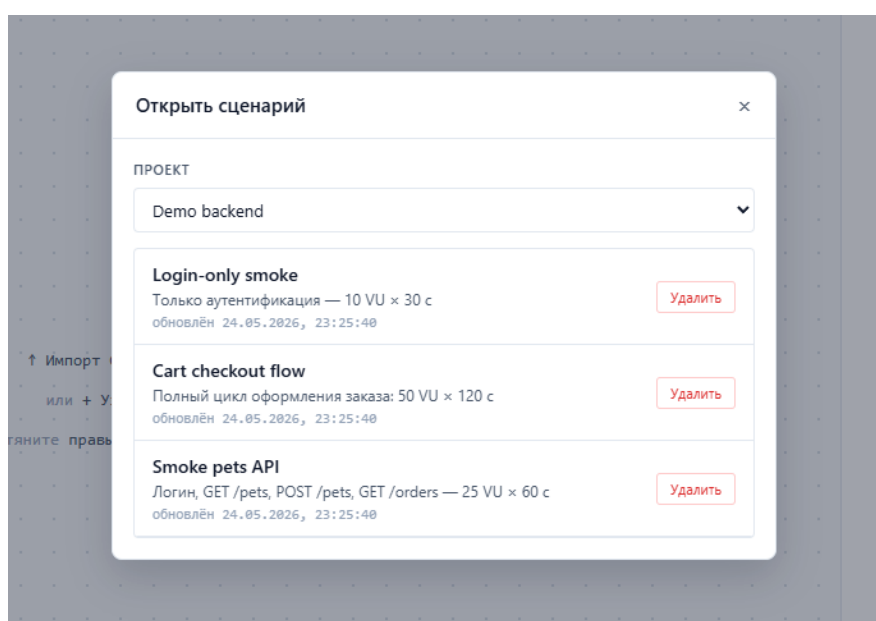


Рисунок 5.9 – Диалог открытия ранее сохраненного окна

Внешний вид редактора при работе с загруженным сценарием представлен на рисунке 5.10. Окно редактора состоит из четырёх областей: верхней панели инструментов, левой клетчатой рабочей области (холста), на которой размещаются узлы сценария и связи между ними, правой панели свойств выбранного элемента и нижнего ряда вкладок «Свойства», «YAML», «Сценарий», позволяющих переключаться между визуальным редактором, текстовым представлением сценария в формате YAML и общими настройками сценария.

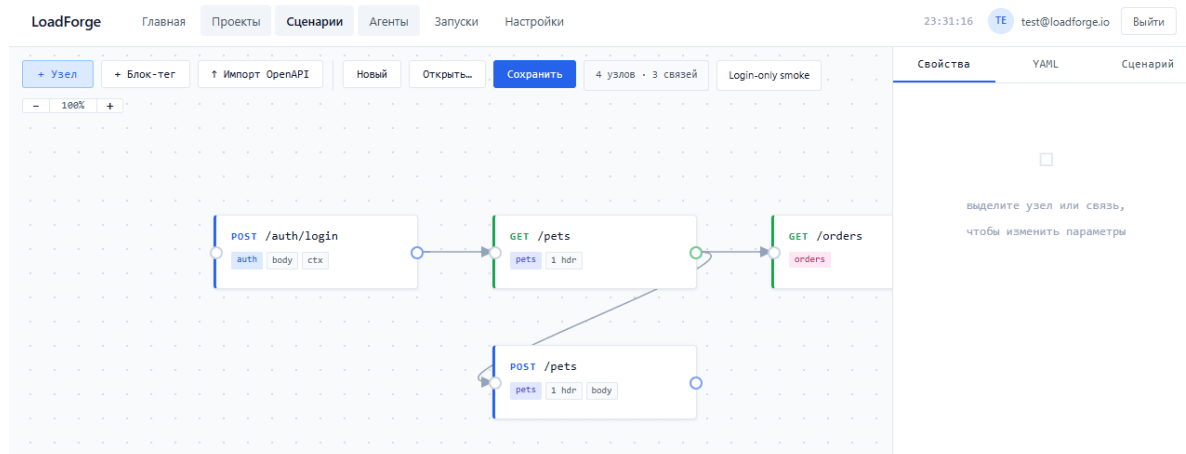


Рисунок 5.10 – Графический редактор с загруженным сценарием

Таким образом, графический редактор объединяет в едином окне визуальное проектирование сценария, его текстовое представление в формате YAML и общие настройки запуска, что позволяет работать с одним и тем же сценарием как в режиме «no-code», так и с прямым редактированием исходного описания. Дальнейшие подразделы описывают типовые способы наполнения сценария содержимым – от автоматического импорта существующей спецификации до ручной сборки шагов из примитивных узлов.

5.9.1 Импорт OpenAPI-спецификации

Наиболее быстрый способ создания тестового сценария – импорт OpenAPI-спецификации тестируемой системы. Для этого следует нажать кнопку «Импорт OpenAPI» на верхней панели редактора и выбрать в стандартном диалоге браузера файл спецификации в формате JSON или YAML, поддерживающий стандарт OpenAPI 3.x. Платформа разбирает спецификацию, группирует операции по их первому тегу и размещает на холсте узлы для каждого эндпоинта, объединённые в блоки-теги. Такая группировка облегчает восприятие крупных API, в которых число операций может исчисляться десятками, и позволяет пользователю быстро находить

нужные эндпоинты по функциональному признаку. При создании узлов платформа автоматически переносит из спецификации описание параметров запроса, заголовков и схемы тела, что освобождает пользователя от необходимости вручную воспроизводить структуру обращений к API и существенно сокращает время подготовки сценария. После импорта пользователю достаточно соединить узлы в требуемой логической последовательности при помощи операций перетаскивания, описанных в подразделе 5.9.4, и при необходимости откорректировать параметры конкретных шагов в правой панели свойств. Импорт можно выполнять повторно при обновлении спецификации тестируемой системы – платформа добавит на холст новые эндпоинты, не затрагивая ранее настроенные связи между существующими узлами. Подобный подход обеспечивает плавную эволюцию сценариев совместно с тестируемой системой и устраняет необходимость в полном пересоздании сценария при расширении публичного контракта API.

5.9.2 Графический редактор шагов

Узлы сценария образуют двух типов: узел шага, представляющий собой одиночный HTTP-запрос (метод и путь), и блок-тег, объединяющий несколько узлов-шагов по логической принадлежности. Для добавления нового узла следует нажать кнопку «+ Узел» на верхней панели; для добавления нового блока-тега – кнопку «+ Блок-тег». Перемещение узлов по холсту выполняется операцией перетаскивания. Масштаб холста регулируется кнопками «–» и «+» в верхней левой части окна или прокруткой колеса мыши при удерживаемой клавише Ctrl. Удаление узла или связи между узлами выполняется выделением соответствующего элемента и нажатием клавиши Delete на клавиатуре.

5.9.3 Свойства узла

Для редактирования параметров узла-шага следует выделить его одиночным нажатием левой кнопки мыши; в правой панели «Свойства» отображаются поля: строка запроса (формат «МЕТОД /путь», например, GET /api/v1/users/{id}), теги (для последующей группировки и условных переходов), заголовки запроса (в виде списка пар «ключ – значение»), параметры строки запроса, path-параметры (отдельная таблица), тело запроса (свободный текст, при необходимости в формате JSON), задержка перед выполнением шага (например, 500ms), правила извлечения значений из ответа в контекст виртуального пользователя (на основе gjson-выражений, формат «gjson-выражение → имя переменной»). Поля свойств шага сохраняются в

редактируемом сценарии немедленно по мере ввода; явного действия «Сохранить» для каждого поля не требуется.

5.9.4 Условные переходы между шагами

Связь между двумя узлами создаётся операцией перетаскивания: следует навести указатель мыши на правый круглый порт исходного узла и при удерживаемой левой кнопке мыши перенести курсор на левый круглый порт целевого узла. Связь между двумя узлами соответствует условному переходу: после выполнения исходного шага агент может перейти к выполнению целевого шага в случае выполнения условия, заданного для этой связи. Условие перехода задаётся в правой панели свойств выделенной связи: пользователь указывает HTTP-статусы ответа (одно или несколько точных значений, либо маски формата 2xx и 4xx), при которых данный переход активируется, и при необходимости задаёт извлечение значений из тела ответа и заголовков ответа в переменные контекста, доступные на последующих шагах. Такой механизм позволяет реализовывать в сценарии передачу состояния между шагами – например, использование полученного при аутентификации токена в последующих запросах – без написания вспомогательного кода. При наличии нескольких подходящих переходов из одного шага выбор между ними выполняется в соответствии со стратегией, заданной в свойствах сценария: *random* – случайный выбор, *sequential* – последовательный циклический обход. При отсутствии явных переходов следующий шаг выбирается в линейном порядке, заданном в сценарии. Если же ни один из заданных переходов не соответствует фактическому ответу сервера, исполнение текущей виртуальной сессии прекращается, а результат шага фиксируется в статистике запуска как несостоявшийся переход, что упрощает последующий анализ некорректно работающих сценариев.

5.9.5 Просмотр YAML-эквивалента и общие настройки сценария

Вкладка «YAML» в правой части окна редактора отображает текстовое представление редактируемого сценария в формате YAML, эквивалентное его внутреннему JSON-представлению, представлен на рисунке 5.11. Представление YAML обновляется в реальном времени по мере внесения изменений в графический редактор и предназначено для предварительного просмотра, отладки и при необходимости – копирования содержимого для запуска агента в автономном режиме командой `loadforge-agent run`. Дополнительно над текстом представлены кнопки «копировать» и «скачать», позволяющие быстро поместить содержимое в буфер обмена или сохранить

его в файл. Полученный таким образом файл сценария представляет собой самостоятельный артефакт, который удобно хранить в системе контроля версий рядом с исходным кодом тестируемого приложения и использовать для воспроизведения нагрузочных тестов в автоматизированных конвейерах сборки. Такой подход обеспечивает прозрачность изменений в составе тестовых сценариев и упрощает их совместное сопровождение командами разработки и тестирования.

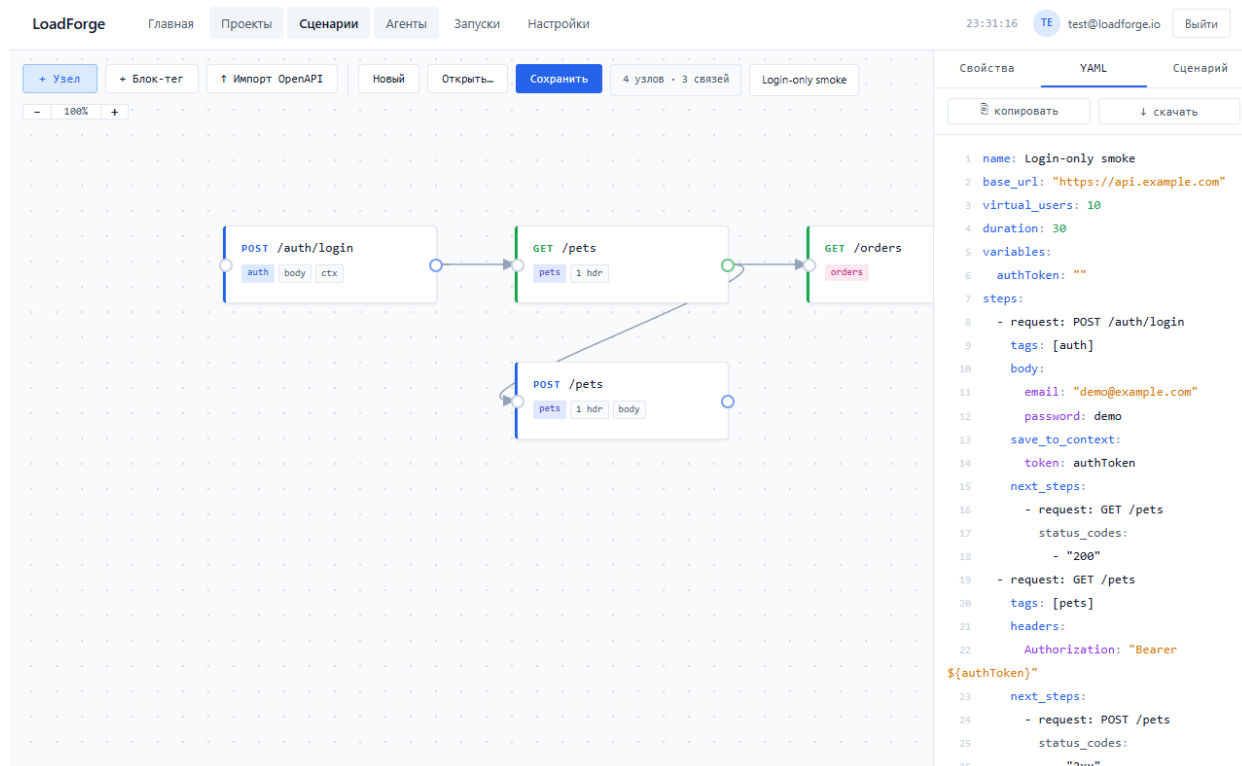


Рисунок 5.11 – Просмотр сценариев в виде YAML

Текстовое представление, показанное на рисунке 5.11, повторяет логическую структуру сценария и состоит из двух смысловых частей. В заголовочной части задаются общие параметры запуска: имя сценария, базовый адрес тестируемой системы, число виртуальных пользователей, длительность теста и набор глобальных переменных. За ней следует список шагов, каждый узел графа – это отдельный шаг, описываемый строкой запроса вида «МЕТОД /путь» и при необходимости дополняемый заголовками, параметрами строки запроса, путевыми параметрами и телом запроса, а также правилами сохранения значений из ответа в контекст и перечнем условных переходов к последующим шагам. Тем самым YAML-описание является точным текстовым отражением графа, построенного на полотне: каждой связи между узлами отвечает отдельный переход в составе соответствующего шага, а порядок шагов задаёт линейный маршрут исполнения по умолчанию.

Вкладка «Сценарий» в правой части окна редактора содержит общие настройки тестового сценария: имя сценария, базовый URL тестируемой системы, число виртуальных пользователей, длительность запуска и список глобальных переменных, изображено на рисунке 5.12. Изменения общих настроек применяются к сценарию незамедлительно и отражаются как в графическом представлении, так и в YAML-эквиваленте.

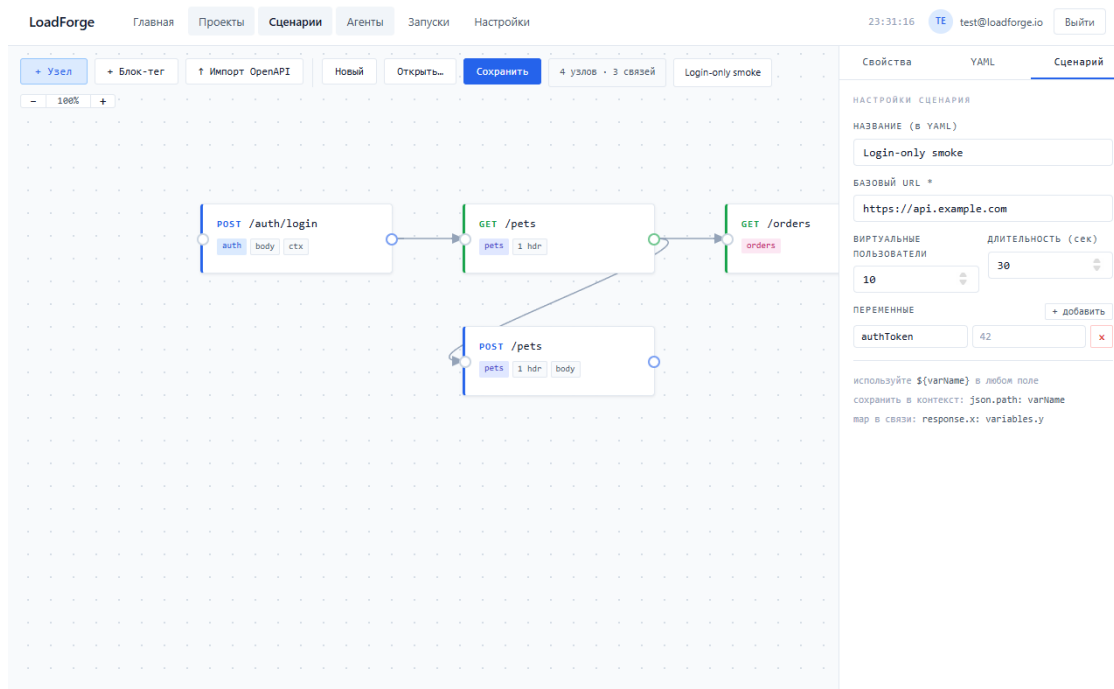


Рисунок 5.12 – Общие настройки тестового сценария

5.9.6 Сохранение и загрузка сценария

Для сохранения сценария на платформу следует нажать кнопку «Сохранить...» на верхней панели редактора, ввести имя сценария, выбрать проект из выпадающего списка и подтвердить сохранение. Сохранённый сценарий немедленно появляется в списке сценариев соответствующего проекта и становится доступен для постановки запуска. Для загрузки ранее сохранённого сценария следует нажать кнопку «Открыть...» и выбрать сценарий из выпадающего списка соответствующего проекта, представленного на рисунке 5.9; редактор полностью восстанавливает состояние холста, включая позиции узлов, связи между ними и значения свойств. Восстановление выполняется без потерь, поскольку сценарии хранятся на платформе в каноническом JSON-формате, однозначно описывающем как структуру графа, так и параметры каждого шага. Для создания нового пустого сценария следует нажать кнопку «Новый» на верхней панели редактора; перед очисткой холста платформа предлагает сохранить текущий черновик, если в

нѐм имеются несохранѐнные изменения. Это исключает случайную потерю результатов работы при переключении между сценариями в рамках одного сеанса редактирования.

5.10 Раздел «Запуски»

Раздел «Запуски» содержит постраничный список всех выполненных и текущих тестовых запусков в рамках организации пользователя. Внешний вид раздела представлен на рисунке 5.13. Каждая строка списка включает: индикатор статуса запуска (одно из значений: «Ожидает», «Выполняется», «Завершѐн», «Ошибка», «Отменѐн»), имя тестируемого сценария, имя агента, выполняющего или выполнившего запуск, продолжительность и время начала выполнения.

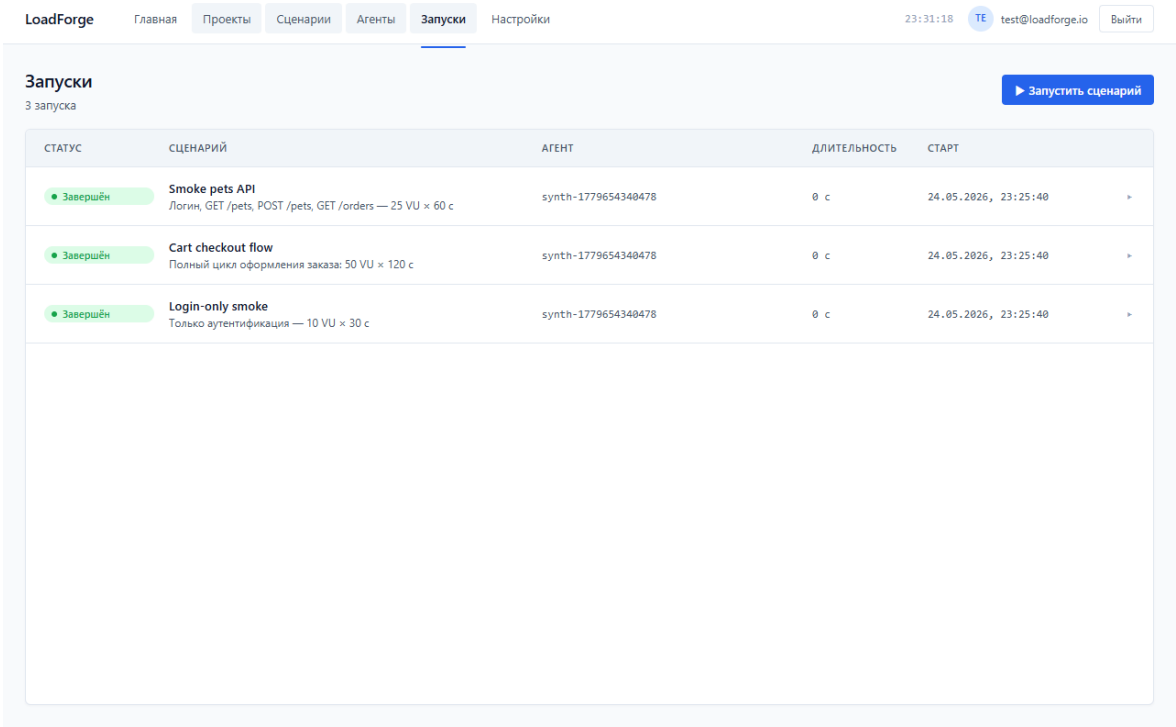


Рисунок 5.13 – Раздел «Запуски»

Для постановки нового запуска следует нажать кнопку «Запустить сценарий» в правом верхнем углу раздела. В открывшемся диалоге пользователю предлагается последовательно выбрать проект, тестовый сценарий в рамках выбранного проекта и агент из числа агентов, прошедших регистрацию. Внешний вид диалога представлен на рисунке 5.14. Все выпадающие списки заполняются автоматически на основании данных текущей организации. Кнопка «Запустить» становится активной, как только все три поля заполнены; после её нажатия запуск ставится в очередь со

статусом «Ожидает». Если в течение нескольких секунд агент, выполнявший к этому моменту длинный опрос, забирает работу, статус автоматически меняется на «Выполняется».

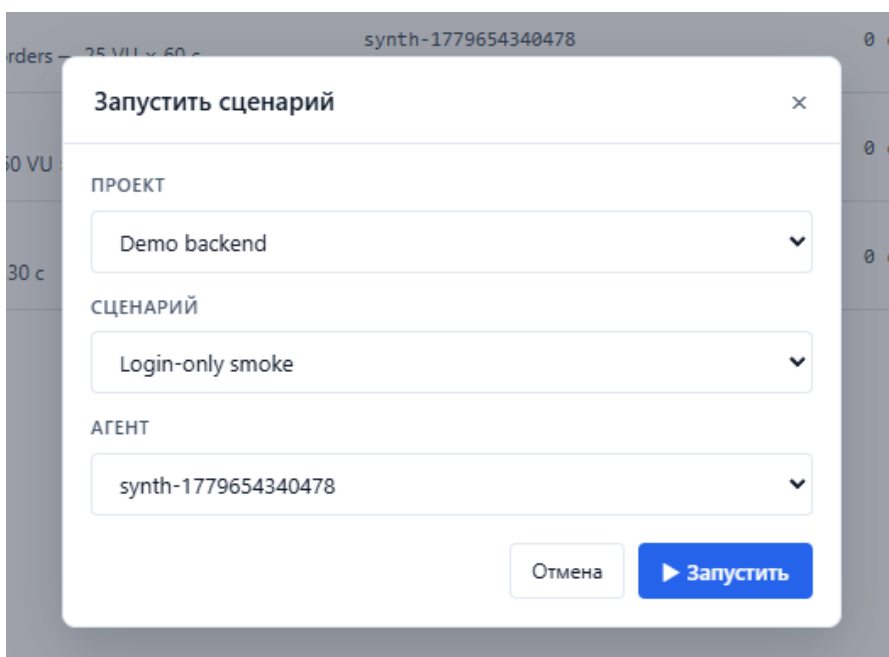


Рисунок 5.14 – Диалог постановки нового тестового запуска

По завершении тестового сценария – либо штатно по истечении заданной в сценарии длительности, либо при получении сигнала остановки – статус запуска переходит в одно из финальных значений: «Завершён» при успешном завершении, «Ошибка» при возникновении неустранимой ошибки в ходе выполнения, «Отменён» при принудительной остановке. Финальный статус фиксируется агентом и передаётся на платформу одним подтверждающим запросом, после чего изменению не подлежит, что обеспечивает однозначную интерпретацию результатов при последующем анализе. Для каждого запуска становится доступна агрегированная сводка результатов, формируемая платформой на основе метрик, собранных агентом в течение всего времени выполнения сценария.

5.11 Просмотр и анализ результатов запуска

Для просмотра подробной сводки по запуску следует нажать на соответствующую строку в списке раздела «Запуски». Строка раскрывается, и под ней отображается панель деталей, включающая агрегированные показатели запуска, таблицу с разбивкой по эндпоинтам и таблицу последних 100 ошибок, с их описанием. В составе агрегированных показателей выводится общее число выполненных запросов, средняя достигнутая

интенсивность нагрузки, перцентили времени отклика, а также доля неуспешных ответов, что даёт пользователю возможность за одно обращение к интерфейсу оценить как общую пропускную способность тестируемой системы, так и стабильность её работы под нагрузкой. Такое представление позволяет получить полную картину результатов теста без перехода на отдельную страницу и упрощает сравнение нескольких запусков между собой. Таблица ошибок, в свою очередь, содержит код ответа, эндпоинт и фрагмент тела ответа, что облегчает локализацию причин отказов и сокращает время на воспроизведение проблемы в среде разработки. Внешний вид раскрытой строки запуска представлен на рисунке 5.15.

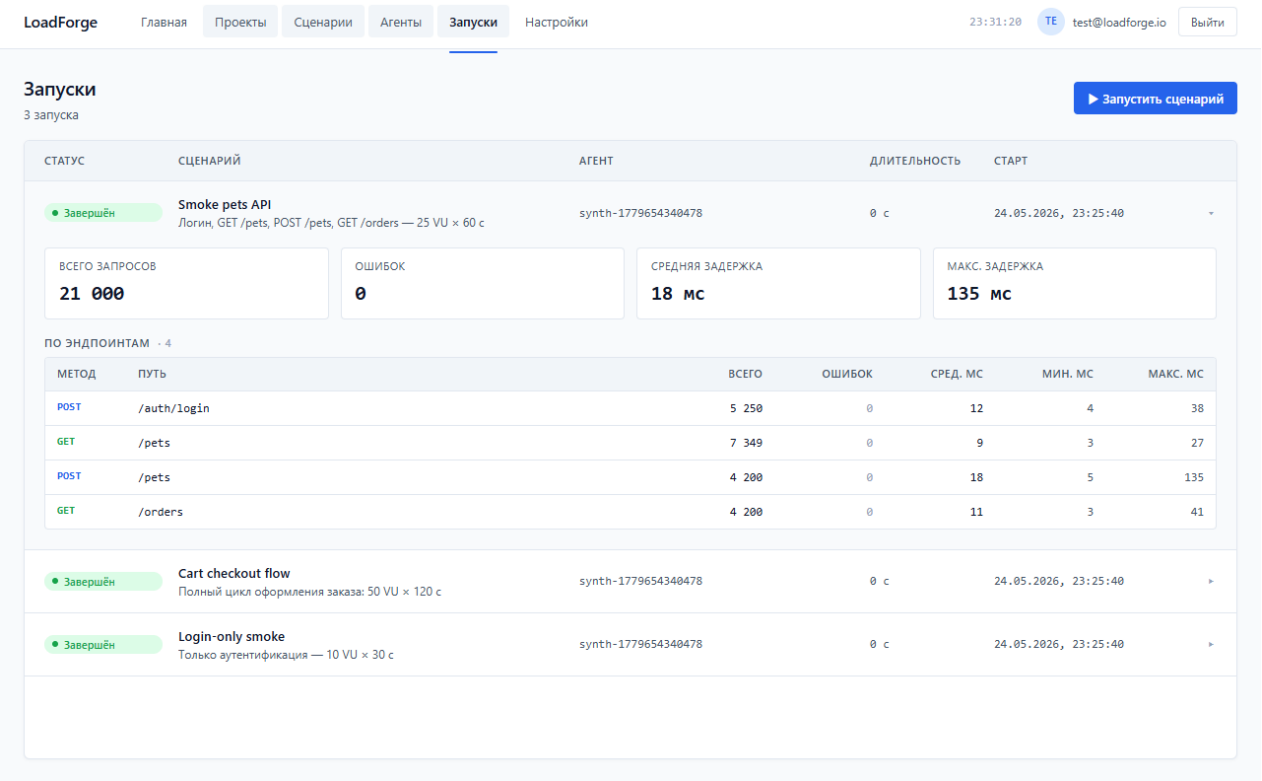


Рисунок 5.15 – Подробная сводка по выполненному запуску

Подробная сводка, показанная на рисунке 5.15, формируется на основе итогового отчёта, который агент передаёт управляющей платформе по завершении запуска, и становится доступной после перехода запуска в завершающее состояние – «Завершён» при успешном прохождении сценария либо «Ошибка» при неуспешном.

Любой из ранее выполненных запусков может быть проведён повторно с теми же параметрами через диалог постановки запуска. Это позволяет оценивать влияние внесённых в тестируемую систему изменений на её производительность и подтверждать воспроизводимость полученных результатов в ходе нескольких последовательных испытаний.

Верхний ряд карточек панели деталей отражает основные показатели запуска: общее число выполненных запросов, число ошибочных запросов, среднюю задержку ответа в миллисекундах и максимальную задержку ответа в миллисекундах. Подобная компоновка обеспечивает мгновенное визуальное восприятие итоговых характеристик нагрузочного теста и позволяет пользователю на первом же экране оценить успешность его выполнения без необходимости погружения в детальные таблицы. Под рядом карточек выводится таблица «По эндпоинтам», содержащая аналогичные показатели в разбивке по конкретным эндпоинтам, использованным в сценарии. Такое представление облегчает выявление наиболее медленных или нестабильных операций тестируемой системы и даёт возможность сопоставить характеристики отдельных шагов сценария между собой – например, обнаружить эндпоинт, на который приходится непропорционально большая доля ошибок при сравнимой интенсивности обращений. В нижней части панели деталей отображается таблица «Последние ошибки», содержащая до ста последних ошибочных запросов с указанием времени, метода, пути, кода ответа, задержки ответа и краткого описания ошибки; таблица отсортирована по убыванию времени, что позволяет в первую очередь видеть свежие проявления проблемы. Ограничение в сто записей выбрано в качестве компромисса между информативностью и нагрузкой на интерфейс: при большем числе ошибок полная их выборка может быть получена через REST API платформы или через выгрузку результатов запуска в виде отдельного файла. При наличии у запуска статуса «Ошибка» в верхней части панели деталей дополнительно выводится текстовое описание причины фатальной ошибки, помешавшей выполнению сценария. К таким причинам относятся, в частности, недоступность тестируемого хоста на этапе инициализации запуска, ошибки разбора сценария на стороне агента и принудительная остановка запуска пользователем. Содержимое раздела «Запуски» автоматически обновляется каждые пять секунд, что обеспечивает наблюдение за выполняющимися запусками в режиме, близком к реальному времени, без необходимости ручного обновления страницы. По завершении запуска автоматическое обновление сохраняется, что позволяет сразу зафиксировать в интерфейсе итоговые агрегированные показатели и переход статуса запуска в одно из терминальных состояний.

5.12 Раздел «Настройки»

Раздел «Настройки» содержит сводную информацию об учётной записи пользователя и параметрах подключения к управляющей платформе. Внешний вид раздела представлен на рисунке 5.16. Раздел состоит из пяти панелей:

«Профиль», «Организации», «Подключение агента», «О приложении» и «Сеанс».

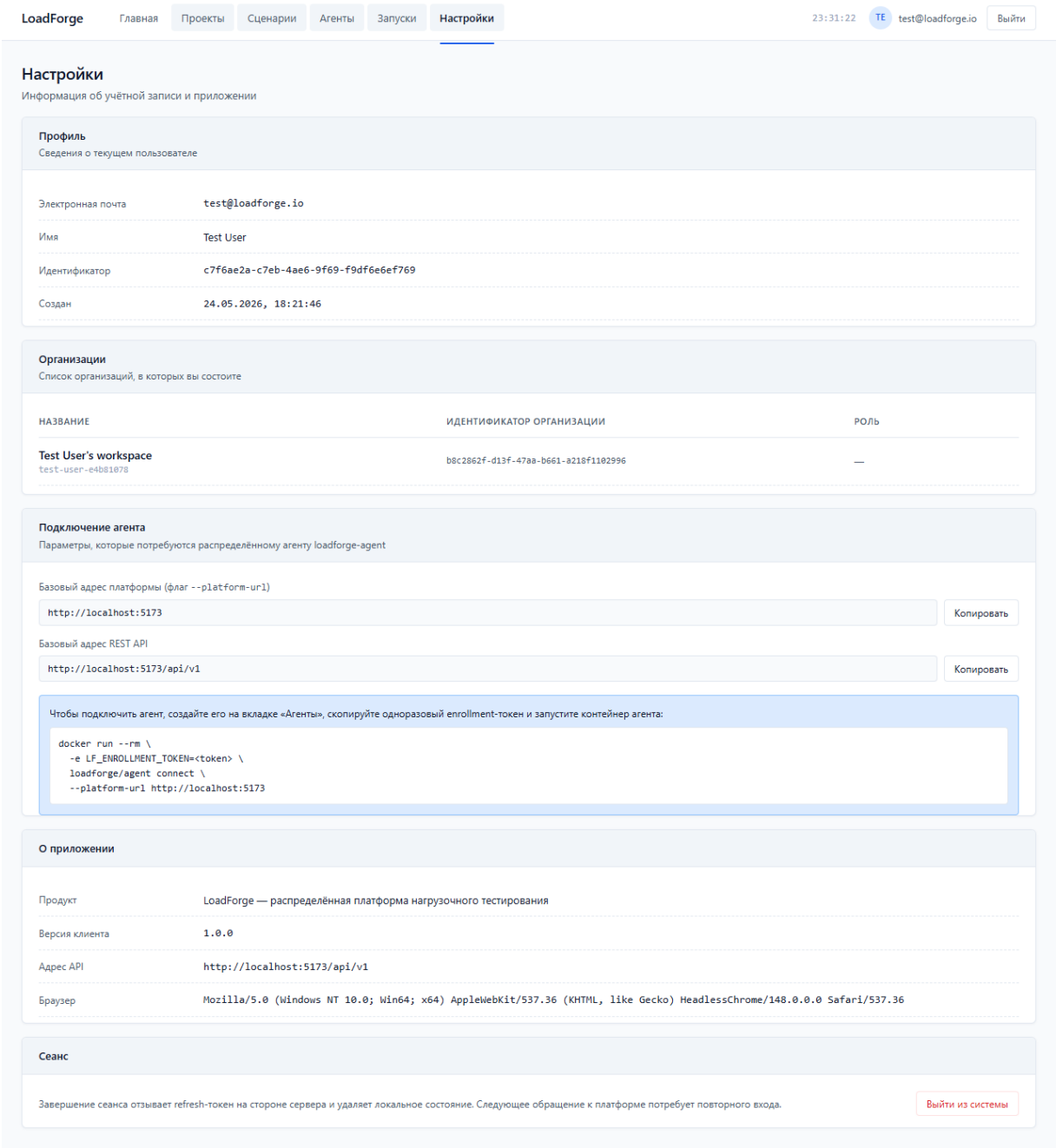


Рисунок 5.16 – Раздел «Настройки»

Панель «Профиль» отображает электронную почту пользователя, его имя, внутренний идентификатор и дату создания учётной записи. Панель «Организации» содержит таблицу организаций, в которых состоит пользователь, с указанием названия, идентификатора и роли. Панель «Подключение агента» содержит копируемые поля с базовым адресом платформы и базовым адресом REST API, а также готовую к копированию команду запуска агента в среде Docker – пользователю достаточно подставить

в неё одноразовый enrollment-токен, скопированный из раздела «Агенты». Панель «О приложении» содержит сведения о продукте, текущей версии клиентского приложения, адресе REST API и строке User-Agent браузера, в котором приложение выполняется. Панель «Сеанс» содержит кнопку «Выйти из системы», нажатие которой отзывает refresh-токен на стороне сервера, удаляет локально сохранённое состояние сеанса и возвращает пользователя к окну входа.

5.13 Сообщения об ошибках

Все нештатные ситуации, возникающие при обращении клиентского приложения к управляющей платформе, отображаются в виде красных горизонтальных плашек с описанием ошибки в верхней части соответствующего раздела. Характерные сообщения и рекомендации по их обработке приведены в таблице 5.1.

Таблица 5.1 – Характерные сообщения об ошибках

Сообщение	Возможная причина	Рекомендуемые действия
invalid credentials	Введена неверная пара «электронная почта – пароль»	Проверить правильность ввода; учесть, что пароль чувствителен к регистру
email already in use	Электронная почта уже используется в системе	Воспользоваться вкладкой «Вход» либо указать другой адрес
unauthorized	Срок действия сеанса истёк	Войти в систему повторно
forbidden	Запрашиваемый ресурс не принадлежит текущей организации	Переключиться на нужную организацию в правом верхнем углу
agent already enrolled	Повторное использование enrollment-токена	Запросить новый токен в разделе «Агенты»
scenario validation failed	Сценарий не прошёл валидацию	Открыть сценарий в редакторе и проверить корректность шагов
no agents available	Не выбран агент или нет агентов «Онлайн»	Зарегистрировать и запустить агента

Завершение работы с системой выполняется нажатием кнопки «Выйти» в правом верхнем углу панели навигации либо кнопки «Выйти из системы» в разделе «Настройки». Обе кнопки отзывают refresh-токен на стороне сервера, удаляют локально сохранённое состояние сеанса и возвращают пользователя к окну входа. После завершения работы рекомендуется также закрыть вкладку браузера во избежание оставления её открытой на устройствах общего пользования.

Таким образом, рассмотренный в настоящем разделе сценарий работы с платформой охватывает полный жизненный цикл нагрузочного тестирования: от регистрации пользователя и создания проекта, через подключение агента и составление тестового сценария, до запуска нагрузочного теста и анализа полученных результатов. Каждый из этапов реализован в виде отдельного раздела веб-интерфейса с единообразной навигацией и согласованным внешним видом, что обеспечивает низкий порог входа для пользователя, впервые сталкивающегося с инструментами нагрузочного тестирования, и одновременно сохраняет необходимую функциональную полноту для решения практических задач инженеров по тестированию производительности.

Особо следует отметить, что предложенная организация пользовательского интерфейса последовательно реализует принятый в проекте принцип разделения управляющей и исполнительной составляющих системы: платформа берёт на себя задачи хранения, отображения и согласования данных между пользователем и распределёнными узлами тестирования, тогда как агенты остаются автономными исполнителями, взаимодействующими с платформой исключительно через защищённый REST-интерфейс. Подобное архитектурное решение, нашедшее отражение в каждом из описанных разделов, обеспечивает горизонтальную масштабируемость инфраструктуры нагрузочного тестирования, гибкость её развёртывания в условиях существующих информационных систем заказчика, а также предсказуемое разграничение зон ответственности между компонентами разрабатываемой системы.

Полученный пользовательский интерфейс, в совокупности с описанными в предшествующих разделах архитектурными и программными решениями, образует целостный программный продукт, готовый к практической апробации в условиях, приближенных к реальным сценариям эксплуатации, описание которой приводится в последующих главах настоящей работы.

6 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

6.1 Характеристика разработанного по индивидуальному заказу программного средства

Разработанное программное средство представляет собой распределённую систему нагрузочного тестирования LoadForge, построенную по архитектуре «control plane – data plane» и предназначенную для использования инженерами по тестированию и разработчиками организации-заказчика, специализирующейся на разработке высоконагруженных веб-сервисов. Система состоит из управляющей платформы, реализованной в виде серверной части на языке Go и клиентского веб-приложения на React, и распределённого агента нагрузочного тестирования, развёртываемого на произвольном числе машин и контейнеров в инфраструктуре заказчика, также написанного на Go.

Целью разработки является автоматизация процесса нагрузочного тестирования внутренних HTTP-сервисов организации, включая создание и хранение тестовых сценариев в декларативном YAML-формате, постановку запусков на распределённых агентах, сбор метрик производительности в режиме реального времени и формирование агрегированной отчётности по результатам выполнения тестов. Программное средство решает задачи сокращения трудоёмкости подготовки и проведения нагрузочных испытаний, унификации формата описания сценариев в рамках команд тестирования и разработки, обеспечения горизонтальной масштабируемости генерации нагрузки и интеграции нагрузочных тестов в конвейеры непрерывной интеграции и доставки.

Архитектура программного средства построена на взаимодействии управляющей платформы и распределённых агентов нагрузки по защищённому протоколу gRPC, что обеспечивает низкую задержку передачи команд и сбор телеметрии в режиме реального времени. Для хранения тестовых сценариев, истории запусков и собранных метрик используется реляционная база данных PostgreSQL, обеспечивающая транзакционную надёжность и поддержку аналитических запросов. Развёртывание системы выполняется в контейнеризованной среде Docker с возможностью оркестрации Kubernetes, что гарантирует горизонтальное масштабирование и совместимость с существующими процессами эксплуатации инфраструктуры заказчика.

Потребность в подобном решении обусловлена ростом сложности тестируемых систем и невозможностью применения существующих

инструментов нагрузочного тестирования без значительных доработок: классические инструменты (Apache JMeter) обладают высоким потреблением ресурсов и ограниченной поддержкой автоматизации; современные фреймворки (Locust, k6) требуют программирования сценариев и не обеспечивают встроенного механизма распределённой оркестрации без привлечения коммерческих облачных платформ. Заказчику необходим лёгкий, специализированный агент, нативно взаимодействующий с собственной управляющей платформой и обеспечивающий полный контроль над инфраструктурой нагрузочного тестирования.

6.2 Расчет затрат на разработку и цена программного средства, созданного по индивидуальному заказу

Процесс разработки программного обеспечения представляет собой сложный и многоэтапный цикл, требующий не только высококвалифицированных специалистов, но и существенных финансовых вложений, что обуславливает необходимость детального анализа всех затратных статей. Только на основе расчетов можно сформировать экономически обоснованную отпускную цену.

Цель данного раздела – рассчитать общие затраты на создание программного средства и определить отпускную цену с учетом нормативной рентабельности.

Одним из способов определения цены разработки программного средства по индивидуальному заказу является способ на основе полных затрат на разработку программного средства организацией-разработчиком. Расчет затрат осуществляется в разрезе следующих статей: основная заработная плата разработчиков; дополнительная заработная плата разработчиков; отчисления на социальные нужды; прочие затраты.

Основная заработная плата представляет собой сумму вознаграждений, выплачиваемых сотрудникам за выполнение должностных обязанностей в рамках проекта. Она рассчитывается как произведение количества отработанных часов каждым специалистом на его почасовую ставку оплаты труда. Размер почасовой ставки определяется на основании квалификации специалиста, его должности и сложившихся на рынке труда показателей оплаты для соответствующей категории работников, что позволяет получить экономически обоснованную оценку затрат на персонал. Кроме того, в соответствии с отраслевыми стандартами, к основной заработной плате добавляются премиальные выплаты, стимулирующие надбавки и иные формы дополнительного материального поощрения. Эти выплаты учитываются с помощью повышающего коэффициента премирования $K_{пр}$. Значение

коэффициента принимается в соответствии с действующими на предприятии локальными нормативными актами и отражает сложившуюся практику материального стимулирования сотрудников за качественное и своевременное выполнение поставленных задач.

Расчет основной заработной платы производится по формуле:

$$Z_o = K_{\text{пр}} \cdot \sum_{i=1}^n Z_{\text{чи}} \cdot t_i, \quad (6.1)$$

где $K_{\text{пр}}$ – коэффициент премий (1,5-2,0);

n – количество исполнителей, занятых разработкой конкретного ПО;

$Z_{\text{чи}}$ – часовая ставка оплаты труда исполнителя i -й категории, бел. руб.;

t_i – трудоемкость работ, выполняемых i -м исполнителем (ч).

Часовой оклад исполнителя определяется путем деления его месячной заработной платы (оклад с надбавками) на среднее количество рабочих часов в месяце, принятое равным 168 ч в соответствии с данными Министерства труда и социальной защиты Республики Беларусь.

Результаты расчета затрат на основную заработную плату команды разработчиков представлены в таблице 6.1.

Таблица 6.1 – Расчет затрат на основную заработную плату команды разработчиков

Категория исполнителя	Месячный оклад, бел. руб.	Часовой оклад, бел. руб.	Трудоёмкость работ, ч	Итого, бел. руб.
Бизнес-аналитик	3000	17,86	20	357
Системный архитектор	4000	23,81	32	762
Backend-разработчик (Go)	3800	22,62	100	2262
Frontend-разработчик (React)	3200	19,05	48	914
QA-инженер	2500	14,88	32	476
DevOps-инженер	3500	20,83	16	333
Итого				5104
Премия и иные стимулирующие выплаты ($K_{\text{пр}} = 1,5$)				2552
Всего затрат на основную заработную плату разработчиков				7656

Итоговая сумма с учетом премиальных выплат составит:

$$З_0 = 5104 \cdot 1,5 = 7656 \text{ бел. руб.}$$

Следовательно, премия и иные стимулирующие выплаты составят:

$$7656 - 5104 = 2552 \text{ бел. руб.}$$

Расчет дополнительной заработной платы можно провести по формуле 6.2. Дополнительная заработная плата включает оплату отпусков, больничных, праздничных дней, сверхурочной работы и прочих выплат, установленных трудовым законодательством. В расчетах используется нормативный коэффициент 15% от основной заработной платы.

$$З_д = \frac{З_0 \cdot Н_д}{100}, \quad (6.2)$$

где $З_0$ – затраты на основную заработную плату, (руб.);

$Н_д$ – норматив дополнительно заработной платы (в нашем случае параметр принят равным 15%).

$$З_д = \frac{7656 \cdot 15}{100} = 1148 \text{ бел. руб.}$$

Расчет отчислений на социальные нужды проводится по формуле 6.3. Данные отчисления включают в себя взносы во внебюджетные фонды: фонд социальной защиты населения, фонд страхования от несчастных случаев на производстве и иные обязательные платежи, предусмотренные законодательством. Норматив составляет 34,6% от общей суммы основной и дополнительной заработной платы.

$$Р_{соц} = \frac{(З_0 + З_д) \cdot Н_{соц}}{100}, \quad (6.3)$$

где $Н_{соц}$ – норматив отчислений на социальные нужды (34,6%).

$$Р_{соц} = \frac{(7656 + 1148) \cdot 34,6}{100} = 3046 \text{ бел. руб.}$$

Прочие затраты включают амортизационные отчисления, расходы на электроэнергию, аренду офисного помещения и оборудования,

командировочные расходы, расходы на управление и реализацию проекта. Данные затраты включаются в себестоимость разработки в процентах от затрат на основную заработную плату и рассчитываются по формуле (6.4) с нормативом, принятым равным 100 %.

$$З_{пз} = \frac{З_0 \cdot Н_{пз}}{100}, \quad (6.4)$$

где $Н_{пз}$ – норматив отчислений на социальные нужды (34,6%).

$$З_{пз} = \frac{7656 \cdot 100}{100} = 7656 \text{ бел. руб.}$$

Полная сумма затрат на разработку программного обеспечения находится путём суммирования всех рассчитанных статей затрат и представлена в таблице 6.2.

Таблица 6.2 – Затраты на разработку программного обеспечения

Статья затрат	Сумма, бел. руб.
Основная заработная плата команды разработчиков	7 656
Дополнительная заработная плата команды разработчиков	1 148
Отчисления на социальные нужды	3 046
Прочие затраты	7 656
Общая сумма затрат на разработку	19 506

Таким образом, общая сумма затрат на разработку программного средства составила:

$$З_p = 7656 + 1148 + 3046 + 7656 = 19506 \text{ бел. руб.}$$

Цена реализации программного средства, разработанного по индивидуальному заказу, рассчитывается на основе полных затрат на разработку с включением плановой прибыли и налога на добавленную стоимость по формуле (6.5). Подобный подход к ценообразованию, известный как метод полных затрат, обеспечивает гарантированное возмещение всех понесённых разработчиком расходов и получение запланированного финансового результата по проекту. Размер плановой прибыли определяется исходя из принятой на предприятии нормы рентабельности и согласованных с

заказчиком условий договора, а ставка налога на добавленную стоимость принимается в соответствии с действующим налоговым законодательством Республики Беларусь.

$$Ц = З_p + П + НДС, \quad (6.5)$$

где $З_p$ – затраты на разработку программного средства, бел. руб.;

$П$ – плановая прибыль организации-разработчика, бел. руб.;

$НДС$ – сумма налога на добавленную стоимость, бел. руб.

Плановая прибыль, включаемая в цену, рассчитывается по формуле (6.6) исходя из нормативного уровня рентабельности затрат. Уровень рентабельности принят равным 25 %, что превышает среднюю ставку по новым срочным рублёвым депозитам физических лиц, сложившуюся в Республике Беларусь в начале 2026 года (около 11,5 % годовых), и обеспечивает экономическую целесообразность разработки для организации-разработчика. Указанное превышение над безрисковой доходностью банковских вкладов выполняет роль премии за коммерческий и технологический риски, неизбежно сопровождающие разработку программного обеспечения, и компенсирует более длительный по сравнению с депозитным вложением срок возврата вложенных средств.

$$П = \frac{З_p \cdot У_p}{100}, \quad (6.6)$$

где $У_p$ – запланированный норматив рентабельность (25%).

$$П = \frac{19506 \cdot 25}{100} = 4877 \text{ бел. руб.}$$

Сумма налога на добавленную стоимость по ставке 20 %, установленной законодательством Республики Беларусь, рассчитывается по формуле:

$$НДС = \frac{(З_p + П) \cdot Н_{дс}}{100}, \quad (6.7)$$

где $Н_{дс}$ – ставка налога на добавленную стоимость (20 %).

$$НДС = \frac{(19506 + 4877) \cdot 20}{100} = 4877 \text{ бел. руб.}$$

Цена реализации программного средства составит:

$$\text{Ц} = 19506 + 4877 + 4877 = 29260 \text{ бел. руб.}$$

Таким образом, отпускная цена разработанного программного средства для организации-заказчика составляет 29 260 бел. руб., что соответствует уровню рентабельности 25 % и обеспечивает экономический интерес организации-разработчика к выполнению заказа.

6.3 Оценка экономического эффекта для организации-заказчика

Экономический эффект для организации-заказчика выражается в приросте чистой прибыли, получаемом в результате применения разработанного под заказ программного средства. Согласно методическим указаниям по экономическому обоснованию проекта по разработке программного обеспечения, прирост чистой прибыли определяется как разность между экономией текущих затрат и приростом текущих затрат, связанных с эксплуатацией программного средства, с учётом ставки налога на прибыль.

Основными источниками экономии текущих затрат у организации-заказчика являются:

- 1) сокращение трудозатрат на подготовку и проведение нагрузочных испытаний за счёт переиспользования YAML-сценариев, автоматизированной оркестрации агентов и интеграции тестов в конвейеры непрерывной интеграции и доставки;
- 2) высвобождение части ставок инженеров по тестированию, ранее занятых исключительно вопросами нагрузочного тестирования с применением классических инструментов;
- 3) отказ от платных подписок на коммерческие облачные платформы нагрузочного тестирования, ранее использовавшихся для генерации распределённой нагрузки.

До внедрения разработанного программного средства в организации-заказчике нагрузочным тестированием занимались два инженера по тестированию, работавшие с классическими инструментами (Apache JMeter) и облачной платформой коммерческого провайдера. После внедрения системы LoadForge за счёт унификации формата сценариев, автоматизации оркестрации агентов и интеграции тестов в CI/CD высвобождается одна штатная единица QA-инженера, которая может быть направлена на решение задач функционального и регрессионного тестирования.

Расчёт годового фонда оплаты труда высвобождаемой штатной единицы QA-инженера с учётом коэффициента премирования (1,5), норматива

дополнительной заработной платы (15 %) и норматива отчислений на социальные нужды (34,6 %) представлен в таблице 6.3.

Таблица 6.3 – Расчет экономии по фонду оплаты труда QA-инженера

Показатель	Значение, бел. руб.
Месячный оклад с учётом коэффициента премирования ($2500 \cdot 1,5$)	3 750
Дополнительная заработная плата ($3750 \cdot 15 / 100$)	563
Отчисления на социальные нужды $((3750 + 563) \cdot 34,6 / 100)$	1 492
Итого месячный фонд оплаты труда на 1 ставку	5 805
Годовой фонд оплаты труда на 1 ставку ($5805 \cdot 12$)	69 660
Высвобождаемая доля ставки QA-инженера, %	50
Годовая экономия по фонду оплаты труда QA ($69\,660 \cdot 0,5$)	34 830

Дополнительная экономия достигается за счёт отказа от подписки на коммерческую облачную платформу нагрузочного тестирования стоимостью около 150 USD в месяц, что в пересчёте по среднему курсу Национального банка Республики Беларусь составляет приблизительно 500 бел. руб. в месяц, или 6 000 бел. руб. в год.

Суммарная экономия текущих затрат составит:

$$\mathcal{E}_3 = 34830 + 6000 = 40830 \text{ бел. руб.}$$

При эксплуатации разработанного программного средства возникает прирост текущих затрат, связанный с поддержкой собственной инфраструктуры платформы и распределённых агентов, включающий расходы на серверное оборудование, системы мониторинга, хранилище данных, а также трудозатраты DevOps-инженера на сопровождение системы (около 10 % рабочего времени). Помимо перечисленных статей в состав эксплуатационных расходов также входят затраты на электроэнергию, обеспечение сетевого подключения серверов платформы, продление SSL-сертификатов, регистрацию доменного имени и периодическое резервное копирование базы данных, что в совокупности образует устойчивый комплекс инфраструктурных издержек. Несмотря на наличие указанных текущих расходов, их совокупный объём существенно ниже потенциальных потерь от сбоев производственных информационных систем в случае отсутствия систематического нагрузочного тестирования, что подтверждает

экономическую обоснованность внедрения разработанного программного средства.

Таблица 6.4 – Расчёт прироста текущих затрат на эксплуатацию системы

Статья прироста затрат	Сумма за год, бел. руб.
Инфраструктура (серверы, мониторинг, хранилище)	2 400
Сопровождение DevOps-инженером (10 % ставки с учётом надбавок)	9 750
Итого прирост текущих затрат	12 150

Прирост чистой прибыли организации-заказчика, получаемый в результате применения разработанного программного средства, рассчитывается по формуле (6.8) с учётом ставки налога на прибыль 20 %, установленной для организаций общего режима налогообложения.

$$\Delta\P_{\text{ч}} = (\mathcal{E}_z - \Delta Z_{\text{тек}}) \cdot \left(1 - \frac{H_{\text{п}}}{100}\right), \quad (6.8)$$

где $\mathcal{E}_z$ – экономия текущих затрат, полученная в результате применения программного средства, бел. руб.;

$\Delta Z_{\text{тек}}$ – прирост текущих затрат, связанный с эксплуатацией программного средства, бел. руб.;

$H_{\text{п}}$ – ставка налога на прибыль (20%).

$$\Delta\P_{\text{ч}} = (40830 - 12150) \cdot \left(1 - \frac{20}{100}\right) = 28680 \cdot 0,8 = 22944 \text{ бел. руб.}$$

Таким образом, ежегодный прирост чистой прибыли организации-заказчика, получаемый в результате применения разработанного программного средства, составляет 22 944 бел. руб.

6.4 Расчет показателей эффективности инвестиций в разработку программного средства

Оценка экономической эффективности инвестиций в разработку программного средства производится на основе сравнения единовременных капитальных затрат организации-заказчика, представленных ценой приобретения программного средства у организации-разработчика, и

ежегодного экономического эффекта, получаемого в результате его применения.

Единовременные затраты (инвестиции) организации-заказчика составляют 29 260 бел. руб., ежегодный экономический эффект – 22 944 бел. руб. Поскольку сумма инвестиций превышает сумму годового экономического эффекта, оценка экономической целесообразности инвестиций осуществляется на основе расчёта и анализа следующих показателей: чистого дисконтированного дохода (ЧДД, NPV), срока окупаемости инвестиций (PP) и рентабельности инвестиций (PI).

Так как сравниваются разновременные результаты (экономический эффект) и затраты (инвестиции), необходимо привести их к единому моменту времени – началу расчётного периода. Для этого применяется коэффициент дисконтирования, рассчитываемый по формуле:

$$P_i = \frac{\Delta\Pi_q}{C} \cdot 100\%, \quad (6.9)$$

где E_n – норма дисконта (в долях единицы);

t – порядковый номер года периода реализации инвестиционного проекта

Норма дисконта принята равной 12 %, что превышает среднюю ставку по новым срочным рублёвым депозитам физических лиц, сложившуюся в Республике Беларусь в начале 2026 года (около 11,5 % годовых), и обеспечивает корректное сравнение денежных потоков с альтернативной возможностью их размещения на банковских депозитах.

Расчётный период принят равным четырём годам и включает год выполнения разработки и внедрения программного средства, а также три года его эксплуатации в организации-заказчике. Выбранная длительность расчётного периода соответствует типовому сроку использования программных средств данного класса до их существенной модернизации и одновременно укладывается в горизонт прогнозирования, на котором сохраняется приемлемая точность исходных финансово-экономических допущений. Чистый дисконтированный доход рассчитывается по формуле:

$$\text{ЧДД} = \sum_{t=1}^n (P_t \cdot \alpha_t - Z_t \cdot \alpha_t), \quad (6.10)$$

где P_t – результат (экономический эффект), полученный в году t , бел. руб.;

Z_t – затраты (инвестиции в разработку и внедрение программного средства) в году t , бел. руб.

Расчет показателей эффективности инвестиционного проекта по годам расчетного периода представлен в таблице 6.5

Таблица 6.5 – Расчет показателей эффективности инвестиционного проекта по разработке программного обеспечения

Показатель	1 год	2 год	3 год	4 год
Коэффициент дисконтирования	1,000	0,893	0,797	0,712
Экономический эффект (P_t), бел. руб.	0	22 944	22 944	22 944
Дисконтированный результат, бел. руб.	0	20 489	18 286	16 336
Инвестиции (Z_t), бел. руб.	29 260	0	0	0
Дисконтированные инвестиции, бел. руб.	29 260	0	0	0
Чистый дисконтированный доход по годам, бел. руб.	–29 260	20 489	18 286	16 336
Чистый дисконтированный доход нарастающим итогом, бел. руб.	–29 260	–8 771	9 515	25 851
Рентабельность инвестиций, %	–	–	–	188,3

Анализ данных таблицы 6.5 позволяет проследить динамику окупаемости инвестиционного проекта в течение расчётного периода. На первом году организация-заказчик несёт единовременные капитальные затраты в размере 29 260 бел. руб., связанные с приобретением и внедрением программного средства, что отражается отрицательным значением чистого дисконтированного дохода нарастающим итогом. Начиная со второго года проект приносит ежегодный дисконтированный экономический эффект, постепенно сокращающийся в номинальном выражении за счёт действия коэффициента дисконтирования: с 20 489 бел. руб. во второй год до 16 336 бел. руб. в четвёртый. Несмотря на снижение дисконтированных значений во времени, накопленный экономический эффект уверенно покрывает первоначальные инвестиционные затраты в течение третьего года расчётного периода, что подтверждает устойчивость инвестиционной модели проекта к выбранной норме дисконта.

По итогам расчёта чистый дисконтированный доход за четырёхлетний период составил 25 851 бел. руб., что превышает ноль и свидетельствует об экономической эффективности инвестиций в разработку и применение программного средства.

Срок окупаемости инвестиций определяется как момент времени, когда дисконтированный экономический эффект нарастающим итогом становится равным дисконтированной сумме инвестиций. Из таблицы 6.5 видно, что окупаемость наступает в течение третьего года расчётного периода. Более точное значение срока окупаемости определяется по формуле:

$$PP = 2 + \frac{|ЧДД_2|}{P_3 \cdot \alpha_3}, \quad (6.11)$$

где $|ЧДД_2|$ – модуль отрицательного значения чистого дисконтированного дохода нарастающим итогом по итогам второго года, бел. руб.;

$P_3 \cdot \alpha_3$ – дисконтированный экономический эффект третьего года, бел. руб.

$$PP = 2 + \frac{8771}{18286} = 2,48 \text{ года.}$$

Таким образом, дисконтированный срок окупаемости инвестиций составляет приблизительно 2 года и 6 месяцев. Простой (без учёта дисконтирования) срок окупаемости, определяемый как отношение инвестиций к ежегодному приросту чистой прибыли ($29\,260 / 22\,944 = 1,28$ года), составляет около 1 года и 3 месяцев, что укладывается в принятый расчётный период.

Рентабельность инвестиций определяется как отношение суммы дисконтированных результатов к сумме дисконтированных инвестиций и рассчитывается по формуле:

$$P_{\text{и}} = \frac{\sum P_t \cdot \alpha_t}{\sum Z_t \cdot \alpha_t} \cdot 100\%, \quad (6.12)$$

$$P_{\text{и}} = \frac{55111}{29260} \cdot 100\% = 188,3\%.$$

Рассчитанное значение рентабельности инвестиций превышает 100 %, что подтверждает экономическую целесообразность вложения средств в разработку и применение программного средства по сравнению с

альтернативной возможностью размещения данной суммы на банковских депозитах под среднюю процентную ставку.

Сводные результаты экономического обоснования разработки программного средства представлены в таблице 6.6.

Таблица 6.6 – Сводные технико-экономические показатели проекта

Показатель	Значение
Затраты на разработку программного средства, бел. руб.	19 506
Цена программного средства (для заказчика), бел. руб.	29 260
Уровень рентабельности затрат на разработку, %	25,0
Экономия текущих затрат у заказчика за год, бел. руб.	40 830
Прирост текущих затрат у заказчика за год, бел. руб.	12 150
Прирост чистой прибыли у заказчика за год, бел. руб.	22 944
Чистый дисконтированный доход за 4 года, бел. руб.	25 851
Простой срок окупаемости инвестиций, лет	1,28
Дисконтированный срок окупаемости инвестиций, лет	2,48
Рентабельность инвестиций, %	188,3

На основании выполненных расчётов можно сделать вывод об экономической целесообразности разработки и применения программного средства LoadForge. Чистый дисконтированный доход превышает ноль, рентабельность инвестиций превышает 100 %, срок окупаемости укладывается в расчётный период, что в совокупности подтверждает экономическую обоснованность реализации инвестиционного проекта как для организации-разработчика, так и для организации-заказчика.

7 ОХРАНА ТРУДА

7.1 Производственная санитария, техника безопасности и пожарная профилактика

Пользователи ПЭВМ могут подвергаться воздействию опасных и вредных производственных факторов, таких как повышенные уровни:

- 1) электромагнитного, ультрафиолетового и инфракрасного излучения; статического электричества;
- 2) повышенное или пониженное содержание аэроионов в воздухе рабочей зоны;
- 3) повышенный или пониженный уровень освещенности рабочей зоны, содержание в воздухе рабочей зоны вредных веществ (оксид углерода, озон, аммиак, фенол, формальдегид);
- 4) напряжение зрения, памяти, внимания; длительное статическое напряжение; монотонность труда;
- 5) нерациональная организация рабочего места;
- 6) эмоциональные перегрузки.

Требования к рабочим местам пользователей ПЭВМ проведены в Гигиеническом нормативе «Показатели безопасности и безвредности факторов производственной среды и трудового процесса при работе с видеодисплейными терминалами и электронно-вычислительными машинами» (постановление Совета Министров Республики Беларусь от 25.01.2021 г. № 37, в ред. от 29.11.2022 г. № 829) и Типовой инструкцией по охране труда при использовании в работе офисного оборудования (постановление Министерства труда и социальной защиты от 14.04.2021 № 25).

Площадь помещения на одного пользователя ПЭВМ на базе плоских дискретных экранов (жидкокристаллические, плазменные) должна быть не менее 4,5 м².

7.1.1 Метеоусловия

В производственных помещениях, в которых работа с использованием ПЭВМ является основной (операторские, расчетные, посты управления, залы вычислительной техники), обеспечиваются оптимальные параметры микроклимата, указанные в таблице 7.1. Работа с компьютером относится к категории 1а (работы, производимые сидя и сопровождающиеся незначительным физическим напряжением, при которых расход энергии составляет до 120 ккал/ч, т.е. до 139 Вт). Соблюдение указанных параметров микроклимата способствует поддержанию работоспособности оператора в

течение всей рабочей смены и снижает вероятность возникновения профессиональных заболеваний, связанных с длительным пребыванием в неблагоприятных условиях.

Таблица 7.1 – Оптимальные параметры климата для помещений с ВДТ, ЭВМ и ПЭВМ

Период года	Категория работ	Температура воздуха, °С, не более	Относительная влажность воздуха, %	Скорость движения воздуха, м/с
Холодный	1а	22-24	40-60	0,1
Теплый	1а	23-25	40-60	0,1

Интенсивность теплового излучения работающих от нагретых поверхностей технологического оборудования, осветительных приборов, инсоляции на постоянных рабочих местах не превышает значений, указанных в таблице 7.2.

Таблица 7.2 – Предельно допустимые уровни интенсивности излучения в инфракрасном и видимом диапазоне излучения на расстоянии 0,5 м со стороны экрана ВДТ, ЭВМ и ПЭВМ

Диапазоны длин волн	400-760 нм	760-1050 нм	свыше 1050 нм
Предельно допустимые уровни	0,1 Вт/м ²	0,05 Вт/м ²	4,0 Вт/м ²

Для создания нормальных метеорологических условий наиболее целесообразно уменьшить тепловыделения от самого источника – монитора, что предусматривается при разработке его конструкции.

В производственных помещениях для обеспечения необходимых показателей микроклимата предусмотрены системы отопления, вентиляции и кондиционирования воздуха.

7.1.2 Вентиляция и отопление

Воздух рабочей зоны помещения соответствует санитарно-гигиеническим требованиям по содержанию вредных веществ и частиц пыли, приведенным в ГОСТ 12.1.005 – 88 ССБТ «Общие санитарно-гигиенические требования к воздуху рабочей зоны» и Гигиеническом нормативе «Показатели

безопасности и безвредности микроорганизмов –продуцентов, микробных препаратов и их компонентов, вредных веществ в воздухе рабочей зоны и на кожных покровах работающих» (постановление Совета Министров Республики Беларусь от 25.01.2021 № 37).

В помещениях проводится ежедневная влажная уборка и проветривание после каждого часа работы.

Уровни положительных и отрицательных аэроионов, а также коэффициент униполярности в воздухе всех помещений, где расположены ПЭВМ, соответствуют значениям, указанным в таблице 7.3.

Таблица 7.3 – Аэроионный состав воздуха в зоне дыхания производственных и общественных помещений

Уровни аэроионизации	Концентрация аэроионов, ρ (ион/см ³)	Коэффициент униполярности, $У$
Минимально допустимые	$\rho \llcorner + \gg \geq 400$ $\rho \llcorner - \gg \geq 600$	$0,4 < У < 1,0$

Одним из мероприятий по оздоровлению воздушной среды является устройство систем вентиляции и отопления. Задачей вентиляции является обеспечение чистоты воздуха и параметров метеорологических условий на рабочих местах. Чистота воздушной среды достигается удалением загрязненного или нагретого воздуха из помещения и подачей в него свежего воздуха. Для поддержания нормального микроклимата необходим достаточный объем вентиляции, для чего предусматривается кондиционирование воздуха, осуществляющее поддержание постоянных параметров микроклимата в помещении независимо от наружных условий. В помещении применяется приточно-вытяжная система вентиляции.

Параметры микроклимата поддерживаются в холодный период года за счет системы водяного отопления с нагревом воды до 100°C, а в теплый – за счет кондиционирования с параметрами, требуемыми СН 4.02.03-2019.

7.1.3 Освещение

В помещении при работе с ПЭВМ предусмотрены естественное и искусственное освещение. Естественное освещение на рабочих местах осуществляется через световые проемы, ориентированные преимущественно на север, северо-восток, восток, запад или северо-запад и обеспечивает коэффициент естественной освещенности не ниже 1,5 %.

Для внутренней отделки интерьера помещений используются материалы

с коэффициентом отражения для потолка – 0,7– 0,8; для стен – 0,5– 0,6; для пола – 0,3– 0,5.

Искусственное освещение в помещениях осуществляется системой общего равномерного освещения, а при работе с документами применяется система комбинированного освещения. Освещенность поверхности стола в зоне размещения документов составляет 300-500 люкс. Освещенность поверхности экрана не более 300 люкс согласно СН 2.04.03-2020. Коэффициент запаса для осветительных установок общего освещения принимается равным 1,4, а коэффициент пульсации – не более 5 %. Указанные параметры обеспечивают снижение зрительного утомления оператора при длительной работе с экраном.

7.1.4 Шум

Источниками шума в помещениях, оборудованных ПЭВМ, являются принтеры, множительная техника и оборудование для кондиционирования воздуха, в самих ПЭВМ – вентиляторы систем охлаждения и трансформаторы.

В таблице 7.4 приведены допустимые уровни шума, которые обеспечиваются малошумным оборудованием, применением звукопоглощающих материалов для облицовки помещений, а также различных звукопоглощающих устройств (перегородки и т. п.).

Таблица 7.4 – Предельно-допустимые уровни звука, эквивалентные уровни звука и уровни звукового давления в октавных полосах частот при работе с ВДТ, ЭВМ и ПЭВМ и периферийными устройствами

Категория нормы шума	Уровни звукового давления, дБ, в октавных полосах со среднегеометрическими частотами, Гц									Уровни звука и эквивалентны е уровни звуча, дБА
	31.5	63	125	250	500	1000	2000	4000	8000	
I	86	71	61	54	49	45	42	40	38	50
II	93	79	70	63	58	55	52	50	49	60
III	96	83	74	68	63	60	57	55	54	65
IV	103	91	83	77	73	70	68	66	64	75

7.1.5 Электробезопасность

Помещение вычислительного центра по степени опасности поражения электрическим током относится к помещениям без повышенной опасности. Данная классификация обусловлена отсутствием в помещении токопроводящих полов, повышенной влажности, химически активной среды, высокой температуры воздуха, а также одновременного прикосновения работников к металлическим корпусам электрооборудования и заземлённым металлоконструкциям, что соответствует критериям отнесения к указанной категории согласно нормативным документам.

Основные меры защиты от поражения током – изоляция и недоступность токоведущих частей; защитное заземление ($R_3 = 4 \text{ Ом}$ ГОСТ 12.1.030 – 81). Дополнительно в составе питающей электрической сети предусматривается применение устройств защитного отключения, срабатывающих при появлении тока утечки, а также организационные меры, включающие периодические инструктажи персонала по электробезопасности, контроль состояния изоляции силовых и сигнальных кабелей и регулярный визуальный осмотр электроустановочных изделий, розеточных групп и шнуров электропитания оборудования.

Первая помощь при поражениях электрическим током состоит из двух этапов: освобождение пострадавшего от действия тока и оказание ему доврачебной медицинской помощи. Освобождение пострадавшего от действия тока производится путём отключения соответствующего участка электросети с помощью ближайшего коммутационного аппарата либо, при невозможности такого отключения, путём отделения пострадавшего от токоведущих частей с использованием сухих диэлектрических предметов – резиновых перчаток, деревянной палки или одежды самого пострадавшего, не прикасаясь к открытым участкам его тела. После освобождения пострадавшего от действия электрического тока необходимо оценить его состояние, обратив внимание на наличие сознания, характер дыхания и пульсацию крупных артерий. При отсутствии у пострадавшего сознания, дыхания и сердечной деятельности до прибытия медицинских работников проводятся реанимационные мероприятия – непрямой массаж сердца и искусственное дыхание, выполняемые непрерывно вплоть до восстановления самостоятельных жизненных функций либо передачи пострадавшего бригаде скорой помощи. Во всех случаях поражения электрическим током необходимо вызвать врача независимо от состояния пострадавшего, поскольку отдалённые последствия воздействия электрического тока могут проявиться спустя значительное время после происшествия и привести к нарушениям сердечной деятельности и работы центральной нервной системы.

7.1.6 Излучение

При работе с дисплеем ПЭВМ могут возникать следующие опасные факторы: электромагнитные и электростатические поля, ультрафиолетовое и инфракрасное излучение. Длительное воздействие перечисленных факторов в дозах, превышающих установленные гигиенические нормативы, способно приводить к повышенной зрительной утомляемости пользователя и снижению остроты зрения, что обуславливает необходимость их регулярного инструментального контроля на рабочих местах операторов. Снижение указанных воздействий достигается ограничением продолжительности непрерывной работы за дисплеем и организацией регламентированных перерывов в течение рабочей смены.

Уровни физических факторов на рабочих местах пользователей, создаваемые ПЭВМ и периферийными устройствами, не превышают предельно-допустимые уровни: электромагнитных и электростатических полей, отраженных в таблицах 7.5 и 7.6, а также ультрафиолетового в таблице 7.7. Соответствие фактических значений установленным нормативам обеспечивается за счёт применения современных мониторов с жидкокристаллическими матрицами, рационального размещения рабочих мест относительно источников излучения и проведения периодической аттестации условий труда с привлечением аккредитованных испытательных лабораторий.

Таблица 7.5 – Предельно допустимые уровни электромагнитных полей от экранов ВДТ, ЭВМ, ПЭВМ

Наименование параметра	Предельно-допустимые уровни
Напряженность электрического поля в диапазоне частот: 5 Гц-2 кГц 2-400 кГц	не более 25,0 В/м не более 2,5 В/м
Плотность магнитного потока магнитного поля в диапазоне частот: 5 Гц-2 кГц 2-400 кГц	не более 250 нТл не более 25 нТл
Напряженность электростатического поля	не более 15 кВ/м

Таблица 7.6 – Предельно допустимые уровни электромагнитных полей при работе с ВДТ, ЭВМ, ПЭВМ от клавиатуры, системного блока, манипулятора «мышь», беспроводных системам передачи информации и иных периферийных устройств

Диапазоны частот	0,3-300 кГц	0,3-3 МГц	3-30 МГц	30-300 МГц	0,3-300 ГГц
Предельно допустимые уровни	25 В/м	15 В/м	10 В/м	3 В/м	10 мкВт/см ²

Таблица 7.7 – Предельно допустимые уровни интенсивности излучения в ультрафиолетовом диапазоне на расстоянии 0,5 м со стороны экрана ВДТ, ЭВМ и ПЭВМ

Диапазоны длин волн	200-280 нм	280-315 нм	315-400 нм
Предельно допустимые уровни	не допускается	0,0001 Вт/м ²	0,1 Вт/м ²

Наиболее эффективным и часто применяемым методом защиты от электромагнитных излучений является установка экранов. Экранируют либо источник излучения, либо рабочее место. Часто экран устанавливают непосредственно на монитор.

При работе монитора на экране кинескопа накапливается электростатический заряд, создающий электростатическое поле. При этом персонал, работающий с монитором, приобретают электростатический потенциал. Заметный вклад в общее электростатическое поле вносят электризующиеся от трения поверхности клавиатуры и мыши.

7.1.7 Пожарная безопасность

По взрывопожарной и пожарной опасности помещения и здания для ЭВМ относятся к категории Д согласно ТКП 474-2013, а по степени огнестойкости относятся ко 2 согласно СН 2.02.05-2020.

Для предотвращения распространения огня во время пожара с одной части здания на другую устраивают противопожарные преграды в виде стен, перегородок, дверей, окон. Особое требование предъявляется к устройству и размещению кабельных коммуникаций.

Для ликвидации пожаров в начальной стадии применяются первичные

средства пожаротушения: внутренние пожарные водопроводы, огнетушители типа ОВП-10, ОУ-2, асбестовые одеяла и др. Нормы первичных средств пожаротушения для вычислительных центров приведены в таблице 7.8.

Таблица 7.8 – Примерные нормы первичных средств пожаротушения для вычислительного центра

Помещение	Площадь, м ²	Углекислотные огнетушители ручные	Порошковые огнетушители
Вычислительный центр	100	1	1

7.2 Требования к помещениям для работы с ВДТ, ЭВМ и ПЭВМ

Помещения для эксплуатации ЭВМ и ПЭВМ имеют естественное и искусственное освещение.

Запрещается выполнение основной работы с использованием ЭВМ и ПЭВМ на постоянных рабочих местах без естественного освещения, если это не обусловлено технологическим процессом.

Естественное освещение на рабочих местах с ЭВМ и ПЭВМ осуществляется через световые проемы, ориентированные преимущественно на север, северо-восток, восток, запад или северо-запад и обеспечивает коэффициент естественной освещенности не ниже 1,5 %. Оконные проемы оборудованы регулирующими устройствами типа жалюзи, занавесей, внешних козырьков и другое.

Площадь одного рабочего места для пользователей ЭВМ и ПЭВМ на базе электронно-лучевой трубки составляет не менее 6 м². Минимальная площадь одного рабочего места для взрослых пользователей и обучающихся учреждений образования с использованием ЭВМ или ПЭВМ на базе ЭЛТ может составлять не менее 4,5 м² при условиях: отсутствия на рабочем месте периферийных устройств (принтер, сканер и другое); продолжительности работы – не более 4 часов в день.

Площадь одного рабочего места для пользователей ЭВМ и ПЭВМ на базе плоских дискретных экранов (жидкокристаллические, плазменные и другое) составляет не менее 4,5 м².

Помещения, где размещаются рабочие места с ЭВМ и ПЭВМ, оборудованы защитным заземлением (защитным занулением) в соответствии с техническими требованиями по эксплуатации. Запрещается размещать рабочие места с ЭВМ и ПЭВМ на расстоянии менее 10 м от силовых кабелей, вводов и высоковольтных трансформаторов.

Помещения, в которых для работы используются преимущественно ЭВМ и ПЭВМ (операторские, расчетные и т.п.), не граничат с помещениями, в которых уровни шума и вибрации превышают нормируемые значения для данной категории проводимых в них работ и их типа (механические цеха, мастерские и т.п.). Звукоизоляция ограждающих конструкций помещений с ЭВМ и ПЭВМ обеспечивает нормируемые параметры шума в них.

Помещения с ЭВМ и ПЭВМ оборудованы системами отопления, кондиционирования воздуха или эффективной приточно-вытяжной вентиляцией.

Полимерные материалы, используемые для внутренней отделки интерьера помещений с ЭВМ и ПЭВМ, соответствуют требованиям Санитарных норм и правил, устанавливающих критерии гигиенической безопасности полимерных и полимеросодержащих материалов, изделий и конструкций, применяемых в промышленном и гражданском строительстве.

Поверхность пола в помещениях эксплуатации ЭВМ и ПЭВМ ровная, без выбоин, нескользкой, удобной для очистки и влажной уборки, обладает антистатическими свойствами.

Для внутренней отделки интерьера помещений, где расположены ЭВМ и ПЭВМ, использованы диффузно отражающие материалы с коэффициентом отражения для потолка – 0,7-0,8; для стен – 0,5-0,6; для пола – 0,3-0,5.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы спроектирована и реализована распределённая система нагрузочного тестирования веб-приложений LoadForge, построенная по архитектуре «control plane – data plane» и состоящая из двух самостоятельно развёртываемых компонентов – лёгкого автономного агента нагрузочного тестирования и управляющей платформы с веб-интерфейсом. Разработанная система решает поставленные в начале работы задачи и реализует все сформулированные функциональные требования.

В ходе выполнения работы был проведён системный анализ предметной области, включающий обзор эволюции инструментов нагрузочного тестирования и сравнительное исследование трёх наиболее распространённых современных решений – Apache JMeter, Locust и k6. По результатам анализа выявлен ключевой пробел в существующем рынке: отсутствие лёгкого, специализированного агента, способного одновременно обеспечить декларативное описание сценариев, высокую производительность, нативную интеграцию с собственной управляющей платформой и простоту распределённого развёртывания.

На основе сформулированных требований была спроектирована и реализована распределённая архитектура системы. Агент нагрузочного тестирования реализован на языке Go с применением библиотек `sobra`, `gorpkg.in/yaml.v3`, `gjson` и стандартной библиотеки `net/http`; он представляет собой автономный исполняемый компонент, поддерживающий выполнение HTTP-сценариев с параметризацией, условными переходами на основе HTTP-статуса ответа и извлечением значений из тел ответов в контекст виртуального пользователя. Управляющая платформа реализована на языке Go в виде трёхслойной архитектуры (`transport` → `service` → `repository`) с применением маршрутизатора `chi`, типизированного слоя доступа к данным `sqlc` на основе встроенной СУБД SQLite и JWT-подсистемы аутентификации; клиентское веб-приложение реализовано на React с собственной системой дизайн-токенов без зависимости от тяжёлых UI-фреймворков.

Взаимодействие агента и управляющей платформы организовано по протоколу HTTP с передачей данных в формате JSON; для назначения работ агентам применён механизм длинного опроса с публикацией событий через внутренний компонент RunBus, обеспечивающий доставку задания агенту в течение единиц десятков миллисекунд после постановки запуска пользователем. Выбор протокола HTTP в качестве транспорта обусловлен его универсальной поддержкой в корпоративных сетях и совместимостью со стандартными средствами сетевой инспекции, что существенно упрощает

развёртывание агентов в инфраструктуре заказчика без необходимости внесения изменений в действующие политики безопасности. Аутентификация агентов основана на двухэтапной модели «одноразовый enrollment-токен → долговременный session-токен» с хранением в базе данных только SHA-256-хешей токенов. Подобная схема обеспечивает разделение этапов первичной регистрации и последующей эксплуатации агента, ограничивает время жизни первичного секрета и исключает возможность восстановления действующих учётных данных при компрометации внутренних таблиц платформы.

Корректность функционирования системы подтверждена комплексным тестированием. Модульные тесты пакетов агента покрывают свыше 90 % операторов в ключевых компонентах исполнения HTTP-запросов, извлечения значений и разбора OpenAPI-спецификаций. Функциональное тестирование всех 27 HTTP-эндпоинтов управляющей платформы подтвердило их соответствие спецификации. Интеграционное тестирование сквозного цикла «пользователь – платформа – агент» прошло успешно. Независимое нагрузочное тестирование инструментом k6 (100 виртуальных пользователей в течение 60 секунд) показало пропускную способность управляющей платформы на уровне 10 378 запросов в секунду при средней задержке ответа 9,5 миллисекунд и 95-м перцентиле 26,85 миллисекунд при практически нулевой доле ошибок – все 470 194 проверки, заложенные в сценарий k6, успешно пройдены.

В соответствии с заданием выполнены: технико-экономическое обоснование разработки программного средства с расчётом затрат, цены, оценкой экономического эффекта для организации-заказчика и показателей эффективности инвестиций; раздел по охране труда, охватывающий производственную санитарию, технику безопасности и требования к помещениям для работы с ВДТ, ЭВМ и ПЭВМ; разработано подробное руководство пользователя, охватывающее все основные операции в веб-интерфейсе платформы и сопровождаемое визуальными примерами интерфейса.

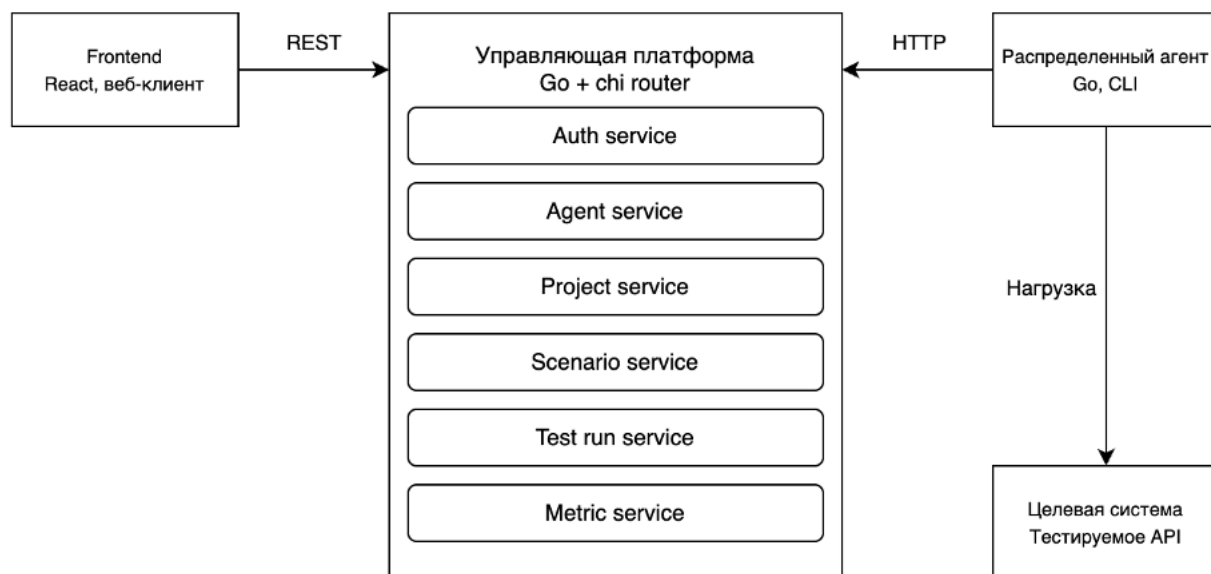
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Donovan A., Kernighan B. The Go Programming Language. – Boston: Addison-Wesley Professional, 2015. – 380 с.
- 2 Effective Go [Электронный ресурс] / The Go Authors. – Режим доступа: https://go.dev/doc/effective_go. – Дата доступа: 21.04.2026.
- 3 The Go Programming Language Specification [Электронный ресурс] / The Go Authors. – Режим доступа: <https://go.dev/ref/spec>. – Дата доступа: 10.04.2026.
- 4 Cobra: A Commander for modern Go CLI interactions [Электронный ресурс]. – Режим доступа: <https://github.com/spf13/cobra>. – Дата доступа: 26.04.2026.
- 5 chi: a lightweight, idiomatic and composable router for building Go HTTP services [Электронный ресурс]. – Режим доступа: <https://github.com/go-chi/chi>. – Дата доступа: 12.05.2026.
- 6 sqlc: Compile SQL queries to type-safe Go [Электронный ресурс]. – Режим доступа: <https://docs.sqlc.dev>. – Дата доступа: 04.04.2026.
- 7 goose: a database migration tool [Электронный ресурс]. – Режим доступа: <https://github.com/pressly/goose>. – Дата доступа: 05.04.2026.
- 8 gjson: Get JSON values quickly from a JSON document [Электронный ресурс] / Т. Holt. – Режим доступа: <https://github.com/tidwall/gjson>. – Дата доступа: 23.05.2026.
- 9 kin-openapi: OpenAPI 3 implementation for Go [Электронный ресурс]. – Режим доступа: <https://github.com/getkin/kin-openapi>. – Дата доступа: 05.05.2026.
- 10 yaml.v3: YAML support for the Go language [Электронный ресурс]. – Режим доступа: <https://gopkg.in/yaml.v3>. – Дата доступа: 07.04.2026.
- 11 RFC 7519: JSON Web Token (JWT) [Электронный ресурс] / М. Jones, J. Bradley, N. Sakimura. – Режим доступа: <https://datatracker.ietf.org/doc/html/rfc7519>. – Дата доступа: 24.04.2026.
- 12 Provos N., Mazières D. A Future-Adaptable Password Scheme // Proceedings of the USENIX Annual Technical Conference. – Monterey: USENIX Association, 1999. – С. 81–91.
- 13 SQLite: a self-contained, serverless, zero-configuration, transactional SQL database engine [Электронный ресурс]. – Режим доступа: <https://www.sqlite.org/docs.html>. – Дата доступа: 08.05.2026.
- 14 OpenAPI Specification [Электронный ресурс] / OpenAPI Initiative. – Режим доступа: <https://spec.openapis.org/oas/v3.0.3>. – Дата доступа: 04.04.2026.
- 15 React: A JavaScript library for building user interfaces [Электронный ресурс] / Meta Platforms, Inc. – Режим доступа: <https://react.dev>. – Дата доступа: 03.05.2026.
- 16 Vite: Next Generation Frontend Tooling [Электронный ресурс]. – Режим доступа: <https://vitejs.dev>. – Дата доступа: 14.04.2026.

- 17 js-yaml: JavaScript YAML parser and dumper [Электронный ресурс]. – Режим доступа: <https://github.com/nodeca/js-yaml>. – Дата доступа: 03.04.2026.
- 18 Recharts: A composable charting library built on React components [Электронный ресурс]. – Режим доступа: <https://recharts.org>. – Дата доступа: 06.04.2026.
- 19 Lucide: Beautiful & consistent icon toolkit made by the community [Электронный ресурс]. – Режим доступа: <https://lucide.dev>. – Дата доступа: 28.04.2026.
- 20 Docker documentation [Электронный ресурс] / Docker, Inc. – Режим доступа: <https://docs.docker.com>. – Дата доступа: 27.04.2026.
- 21 k6: Modern load testing tool, version 0.54.0 [Электронный ресурс] / Grafana Labs. – Режим доступа: <https://k6.io/docs/>. – Дата доступа: 05.04.2026.
- 22 Apache JMeter User's Manual [Электронный ресурс] / The Apache Software Foundation. – Режим доступа: <https://jmeter.apache.org/usermanual/index.html>. – Дата доступа: 16.04.2026.
- 23 Locust: A modern load testing framework [Электронный ресурс] / J. Heyman, C. Byström. – Режим доступа: <https://docs.locust.io>. – Дата доступа: 06.04.2026.
- 24 Newman S. Building Microservices: Designing Fine-Grained Systems. – 2-е изд. – Sebastopol: O'Reilly Media, 2021. – 612 с.
- 25 Fowler M. Patterns of Enterprise Application Architecture. – Boston: Addison-Wesley Professional, 2002. – 533 с.
- 26 Kleppmann M. Designing Data-Intensive Applications. – Sebastopol: O'Reilly Media, 2017. – 616 с.
- 27 Beyer B., Jones C., Petoff J. Site Reliability Engineering: How Google Runs Production Systems. – Sebastopol: O'Reilly Media, 2016. – 552 с.
- 28 Tanenbaum A.S., van Steen M. Distributed Systems: Principles and Paradigms. – 3-е изд. – Upper Saddle River: Pearson, 2017. – 596 с.
- 29 Fielding R. Architectural Styles and the Design of Network-based Software Architectures: дис. ... д-ра философии. – Irvine: University of California, 2000. – 180 с.
- 30 RFC 9110: HTTP Semantics [Электронный ресурс] / R. Fielding, M. Nottingham, J. Reschke. – Режим доступа: <https://datatracker.ietf.org/doc/html/rfc9110>. – Дата доступа: 06.05.2026.
- 31 СанПиН №59 2013. Требования при работе с видеодисплейными терминалами и электронно-вычислительными машинами. – Минск, 2013.
- 32 СанПиН №33 2013. Гигиенические требования к микроклимату производственных и офисных помещений. – Минск, 2013.
- 33 СанПиН №115 2011. Шум на рабочих местах, в транспортных средствах, в помещениях жилых, общественных зданий и на территории жилой застройки. – Минск, 2011.

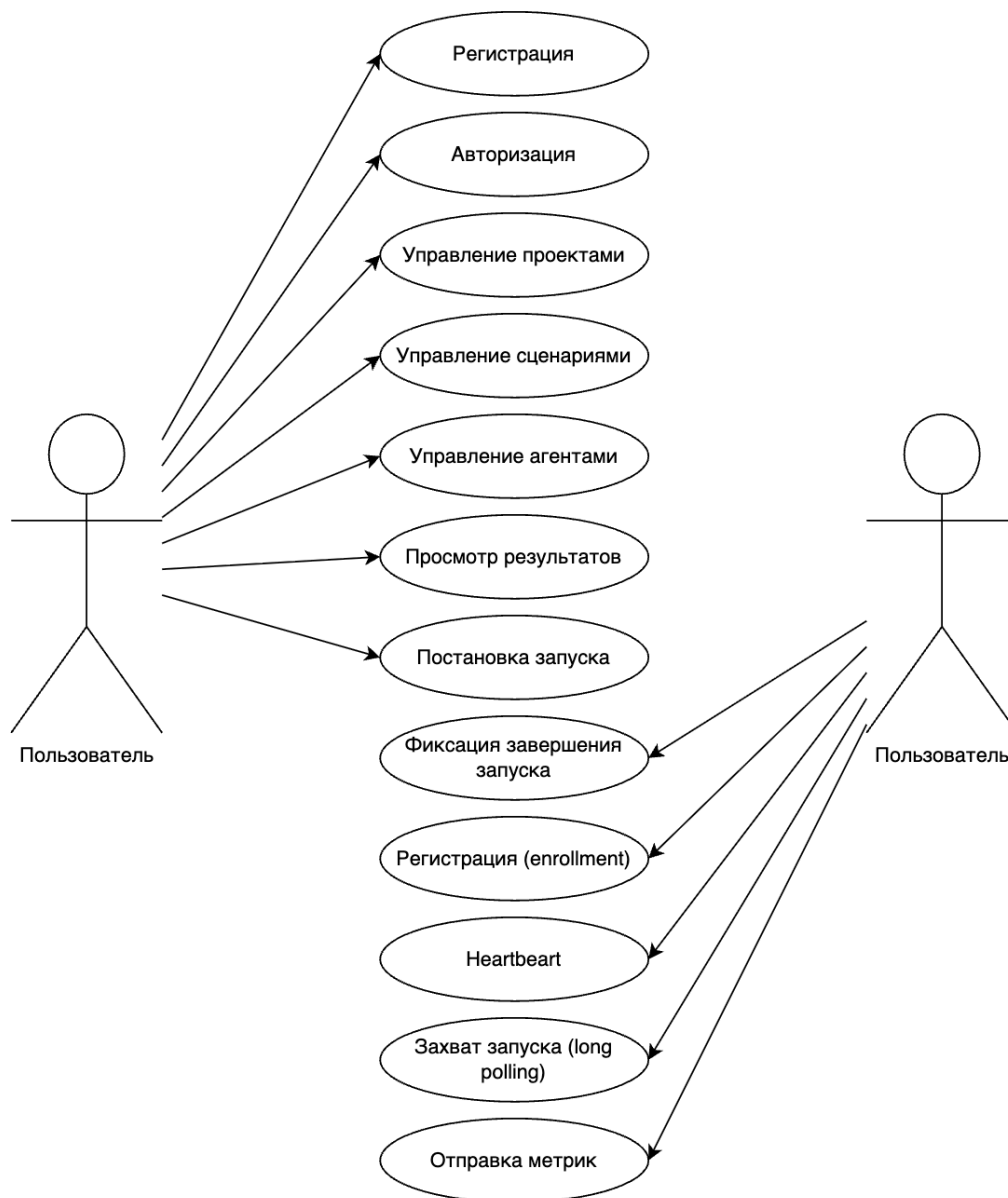
- 34 ТКП 45-2.04-153-2009 Естественное и искусственное освещение. Строительные нормы проектирования.
- 35 ТКП 339-2011 Правила устройства электроустановок.
- 36 ГОСТ 12.2.032-78 ССБТ. Рабочее место при выполнении работ сидя.
- 37 ТКП 474-2013 Категорирование зданий, помещений и наружных установок по взрывопожарной и пожарной опасности.
- 38 СН 2.02.05-2020 Пожарная безопасность зданий и сооружений. Строительные нормы проектирования.
- 39 Лазаренков, А. М. Охрана труда: учебно-методическое пособие / А. М. Лазаренков, И. Н. Ушакова. – Минск: БНТУ, 2011. – 205 с.

Схема взаимодействия компонентов проектируемой системы



					ДП-10706122/22-2026-01			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	1	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Диаграмма вариантов использования



					ДП-10706122/22-2026-02				
Изм.	Лист	№ документа	Подпись	Дата					
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов	
Руковод.		Станкевич				У	2	10	
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск			
Н.контр.		Домаренко							

Результаты нагрузочного тестирования

Показатель	Значение
Число виртуальных пользователей	100
Длительность теста, с	60
Общее число выполненных HTTP-запросов	626 926
Число ошибочных запросов	1
Доля ошибочных запросов, %	0,000159
Завершённых итераций сценария	156 731
Средняя пропускная способность, RPS	10 378
Средняя задержка ответа, мс	9,51
Медианная задержка ответа, мс	7,42
90-й перцентиль задержки, мс	21,50
95-й перцентиль задержки, мс	26,85
Максимальная задержка, мс	428,33
Объём принятых данных, МБ	168,6
Объём отправленных данных, МБ	202,7
Пройдено проверок (k6 checks)	470 194 из 470 194

					ДП-10706122/22-2026-03			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	3	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Структура проекта

loadforge-agent/

└─ cmd/agent/

└─ internal/

└─ cli/

└─ scenario/

└─ executor/

└─ runner/

└─ platform/

репозиторий агента

точка входа, main

внутренние пакеты агента

CLI на cobra: run, connect, version

YAML, парсер, переменные

HTTP-исполнитель поверх net/http

VirtualUser, Stats, планировщик

HTTP-клиент управляющей платформы

loadforge-platform/

└─ backend/

└─ cmd/server/

└─ internal/

└─ domain/

└─ auth/

└─ transport/http/

└─ service/

└─ repository/sqlite/

└─ migrations/

└─ frontend/

репозиторий управляющей платформы

серверная часть на Go

точка входа сервера

внутренние пакеты бэкенда

сущности предметной области

JWT, bcrypt

chi, handlers, middleware, DTO

бизнес-логика (6 сервисов)

sqlc, доступ к данным

goose-миграции

React SPA, потребитель REST API

deploy/

Dockerfile + docker-compose.yml

					ДП-10706122/22-2026-04			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	4	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Окно регистрации в системе

LoadForge

Создайте рабочее пространство

Вход

Регистрация

Имя

Иван Петров

Эл. почта

you@company.com

Пароль

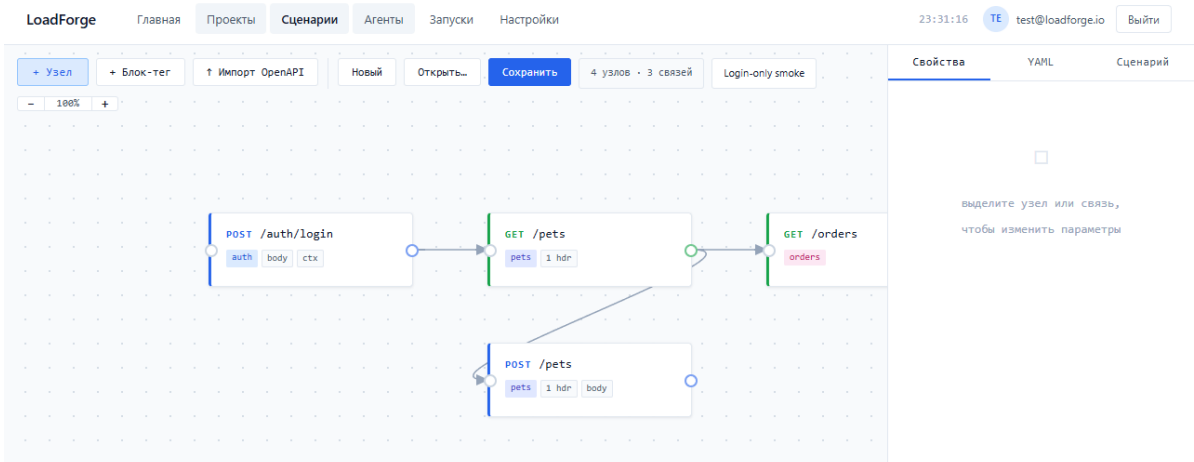
минимум 8 символов

Создать аккаунт

Платформа нагрузочного тестирования · v0.1

					ДП-10706122/22-2026-05			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	5	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Графический редактор сценария



					ДП–10706122/22–2026–06			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк						
Руковод.		Станкевич			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Консульт.		Станкевич				У	6	10
Н.контр.		Домаренко				1-40 01 01		
						БНТУ, г. Минск		

Раздел «Настройки»

LoadForge

Главная

Проекты

Сценарии

Агенты

Запуски

Настройки

23:31:22

TE

test@loadforge.io

Выйти

Настройки

Информация об учётной записи и приложении

Профиль

Сведения о текущем пользователе

Электронная почта

test@loadforge.io

Имя

Test User

Идентификатор

c7f6ae2a-c7eb-4ae6-9f69-f9df6e6ef769

Создан

24.05.2026, 18:21:46

Организации

Список организаций, в которых вы состоите

НАЗВАНИЕ

ИДЕНТИФИКАТОР ОРГАНИЗАЦИИ

РОЛЬ

Test User's workspace

test-user-e4b81878

b8c2862f-d13f-47a8-b661-a218f1102996

—

Подключение агента

Параметры, которые потребуются распределённому агенту loadforge-agent

Базовый адрес платформы (флаг --platform-url)

http://localhost:5173

Копировать

Базовый адрес REST API

http://localhost:5173/api/v1

Копировать

Чтобы подключить агент, создайте его на вкладке «Агенты», скопируйте одноразовый enrollment-токен и запустите контейнер агента:

docker run --rm \

-e LF_ENROLLMENT_TOKEN=<token> \

loadforge/agent connect \

--platform-url http://localhost:5173

О приложении

Продукт

LoadForge — распределённая платформа нагрузочного тестирования

Версия клиента

1.0.0

Адрес API

http://localhost:5173/api/v1

Браузер

Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/148.0.0.0 Safari/537.36

Сеанс

Завершение сеанса отзывает refresh-токен на стороне сервера и удаляет локальное состояние. Следующее обращение к платформе потребует повторного входа.

Выйти из системы

					ДП-10706122/22-2026-07			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	7	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Расчет затрат на основную заработную плату команды разработчиков

Категория исполнителя	Месячный оклад, бел. руб.	Часовой оклад, бел. руб.	Трудоёмкость работ, ч	Итого, бел. руб.
Бизнес-аналитик	3000	17,86	20	357
Системный архитектор	4000	23,81	32	762
Backend-разработчик (Go)	3800	22,62	100	2262
Frontend-разработчик (React)	3200	19,05	48	914
QA-инженер	2500	14,88	32	476
DevOps-инженер	3500	20,83	16	333
Итого				5104
Премия и иные стимулирующие выплаты ($K_{пр} = 1,5$)				2552
Всего затрат на основную заработную плату разработчиков				7656

					ДП-10706122/22-2026-08			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	8	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Сводные технико-экономические показатели проекта

Показатель	Значение
Затраты на разработку программного средства, бел. руб.	19 506
Цена программного средства (для заказчика), бел. руб.	29 260
Уровень рентабельности затрат на разработку, %	25,0
Экономия текущих затрат у заказчика за год, бел. руб.	40 830
Прирост текущих затрат у заказчика за год, бел. руб.	12 150
Прирост чистой прибыли у заказчика за год, бел. руб.	22 944
Чистый дисконтированный доход за 4 года, бел. руб.	25 851
Простой срок окупаемости инвестиций, лет	1,28
Дисконтированный срок окупаемости инвестиций, лет	2,48
Рентабельность инвестиций, %	188,3

					ДП-10706122/22-2026-09			
Изм.	Лист	№ документа	Подпись	Дата				
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит	Лист	Листов
Руковод.		Станкевич				У	9	10
Консульт.		Станкевич				1-40 01 01 БНТУ, г. Минск		
Н.контр.		Домаренко						

Предельно-допустимые уровни звука, эквивалентные уровни звука и уровни звукового давления в октавных полосах частот при работе с ВДТ, ЭВМ и ПЭВМ и периферийными устройствами

Категория нормы шума	Уровни звукового давления, дБ, в октавных полосах со среднегеометрическими частотами, Гц									Уровни звука и эквивалентные уровни звука, дБА
	31.5	63	125	250	500	1000	2000	4000	8000	
I	86	71	61	54	49	45	42	40	38	50
II	93	79	70	63	58	55	52	50	49	60
III	96	83	74	68	63	60	57	55	54	65
IV	103	91	83	77	73	70	68	66	64	75

					ДП-10706122/22-2026-10				
Изм.	Лист	№ документа	Подпись	Дата					
Разраб.		Ханяк			Программное средство нагрузочного тестирования WEB-приложений в изолированной контейнерной среде	Лит		Лист	Листов
Руковод.		Станкевич				У		10	10
Консульт.		Станкевич				1-40 01 01			
Н.контр.		Домаренко				БНТУ, г. Минск			

ПРИЛОЖЕНИЕ А

cmd/agent/main.go

```
package main

import (
    "fmt"
    "os"

    "loadforge-agent/internal/cli"
)

func main() {
    if err := cli.Execute(); err != nil {
        fmt.Fprintln(os.Stderr, err)
        os.Exit(1)
    }
}
```

internal/cli/root.go

```
package cli

import (
    "os"

    "github.com/spf13/cobra"
)

type GlobalFlags struct {
    Verbose    bool
    APIKey     string
    PlatformURL string
}

var globalFlags GlobalFlags

var rootCmd = &cobra.Command{
    Use:     "loadforge",
    Short:   "LoadForge Agent - distributed load testing agent",
    Long:    colorize(colorCyan, banner) + `
LoadForge Agent executes test scenarios defined by LoadForge and reports
metrics back to the control plane.

Source: https://github.com/loadforge/agent`,
    SilenceUsage: true,
    SilenceErrors: true,
}

func Execute() error {
    return rootCmd.Execute()
}

func init() {
    rootCmd.PersistentFlags().BoolVarP(
        &globalFlags.Verbose, "verbose", "v", false,
        "Enable verbose output",
    )
    rootCmd.PersistentFlags().StringVar(
        &globalFlags.APIKey, "api-key", envOrDefault("LOADFORGE_API_KEY", ""),
        "API key for the LoadForge platform (env: LOADFORGE_API_KEY)",
    )
    rootCmd.PersistentFlags().StringVar(
        &globalFlags.PlatformURL, "platform-url",
        envOrDefault("LOADFORGE_PLATFORM_URL", "https://loadforge.io"),
    )
}
```

```

        "LoadForge platform base URL (env: LOADFORGE_PLATFORM_URL)",
    )

    rootCmd.AddCommand(runCmd)
    rootCmd.AddCommand(versionCmd)
}

func envOrDefault(key, def string) string {
    if v := os.Getenv(key); v != "" {
        return v
    }
    return def
}

```

internal/cli/run.go

```

package cli

import (
    "context"
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"

    "github.com/spf13/cobra"

    "loadforge-agent/internal/runner"
    "loadforge-agent/internal/scenario"
)

var runFlags struct {
    dryRun bool
}

var runCmd = &cobra.Command{
    Use:     "run <scenario.yaml>",
    Short:   "Execute a load test scenario",
    Long:    `Run a load test scenario defined in a YAML file.`,
}

```

The scenario file specifies the target base URL, number of virtual users, test duration, and the HTTP steps to execute.

Examples:

```

loadforge run scenario.yaml
loadforge run scenario.yaml --verbose
loadforge run scenario.yaml --api-key=your_key --platform-url=https://loadforge.io
loadforge run scenario.yaml --dry-run`,
    Args: cobra.ExactArgs(1),
    RunE: runScenario,
}

func init() {
    runCmd.Flags().BoolVar(
        &runFlags.dryRun, "dry-run", false,
        "Parse and validate the scenario without executing it",
    )
}

func runScenario(cmd *cobra.Command, args []string) error {
    scenarioFile := args[0]

    if _, err := os.Stat(scenarioFile); os.IsNotExist(err) {
        printError("scenario file not found: %s", scenarioFile)
        return fmt.Errorf("scenario file not found: %s", scenarioFile)
    }
}

```

```

printVerbose("loading scenario from %s", scenarioFile)

parser := scenario.NewParser()
if err := parser.ParseFile(scenarioFile); err != nil {
    printError("failed to parse scenario: %v", err)
    return err
}

if err := parser.Validate(); err != nil {
    printError("scenario validation failed: %v", err)
    return err
}

sc, err := parser.GetScenario()
if err != nil {
    printError("failed to load scenario: %v", err)
    return err
}

printSuccess("scenario loaded: %s", colorize(colorBold, sc.Name))
printInfo("target: %s", sc.BaseURL)
printInfo("virtual users: %d", sc.VirtualUsers)
printInfo("duration: %s", (time.Duration(sc.Duration) *
time.Second).String())
printInfo("steps: %d", len(sc.Steps))

if globalFlags.Verbose {
    for i, step := range sc.Steps {
        printVerbose("  step[%d]: %s", i, step.Request)
    }
}

if globalFlags.APIKey != "" {
    printVerbose("api-key: %s...", globalFlags.APIKey[:min(8,
len(globalFlags.APIKey))])
    printVerbose("platform-url: %s", globalFlags.PlatformURL)
}

if runFlags.dryRun {
    printSuccess("dry-run complete - scenario is valid")
    return nil
}

printInfo("starting load test...")

ctx, cancel := context.WithCancel(context.Background())
defer cancel()

sigCh := make(chan os.Signal, 1)
signal.Notify(sigCh, syscall.SIGINT, syscall.SIGTERM)
go func() {
    sig := <-sigCh
    printWarn("received %s - stopping gracefully...", sig)
    cancel()
}()

r := runner.New(sc)
if err := r.Run(ctx); err != nil {
    printError("load test failed: %v", err)
    return err
}

printSuccess("load test finished")
return nil
}

```

internal/cli/version.go

```

package cli

import (
    "fmt"
    "runtime"

    "github.com/spf13/cobra"
)

var (
    Version    = "dev"
    Commit     = "none"
    BuildDate  = "unknown"
)

var versionCmd = &cobra.Command{
    Use:     "version",
    Short:   "Print version information",
    Long:    `Print the LoadForge Agent version, build commit, and build date.`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Printf("%s\n", colorize(colorCyan+colorBold, "LoadForge Agent"))
        fmt.Printf("  Version:    %s\n", colorize(colorGreen, Version))
        fmt.Printf("  Commit:     %s\n", Commit)
        fmt.Printf("  Built:      %s\n", BuildDate)
        fmt.Printf("  Go version: %s\n", runtime.Version())
        fmt.Printf("  OS/Arch:    %s/%s\n", runtime.GOOS, runtime.GOARCH)
    },
}

```

internal/scenario/scenario.go

```

package scenario

import (
    "fmt"
    "time"
)

type Scenario struct {
    Name           string           `yaml:"name"`
    BaseURL        string           `yaml:"base_url"`
    VirtualUsers   uint64           `yaml:"virtual_users"`
    Duration       uint64           `yaml:"duration"`
    Variables      map[string]string `yaml:"variables,omitempty"`
    Steps          []Step           `yaml:"steps"`
}

type Step struct {
    Request      string           `yaml:"request"`
    Tags         []string         `yaml:"tags,omitempty"`
    Headers      map[string]string `yaml:"headers,omitempty"`
    Query        map[string]string `yaml:"query,omitempty"`
    PathParams   map[string]string `yaml:"path_params,omitempty"`
    Body         interface{}       `yaml:"body,omitempty"`
    Delay        Duration         `yaml:"delay,omitempty"`
    SaveToContext map[string]string `yaml:"save_to_context,omitempty"`
    NextSteps    []NextStep       `yaml:"next_steps,omitempty"`
}

type NextStep struct {
    Request      string           `yaml:"request,omitempty"`
    Tag          string           `yaml:"tag,omitempty"`
    Pick         string           `yaml:"pick,omitempty"`
    StatusCodes []string         `yaml:"status_codes,omitempty"`
    Map          map[string]string `yaml:"map,omitempty"`
}

```

```

func (n NextStep) IsTagTarget() bool { return n.Tag != "" && n.Request == "" }

func (s *Scenario) StepsByTag(tag string) []*Step {
    out := make([]*Step, 0)
    for i := range s.Steps {
        for _, t := range s.Steps[i].Tags {
            if t == tag {
                out = append(out, &s.Steps[i])
                break
            }
        }
    }
    return out
}

type Duration struct {
    time.Duration
}

func (d *Duration) IsZero() bool {
    return d.Duration == 0
}

func (d *Duration) UnmarshalYAML(unmarshal func(interface{}) error) error {
    var raw string
    if err := unmarshal(&raw); err != nil {
        var ns int64
        if err2 := unmarshal(&ns); err2 != nil {
            return fmt.Errorf("delay must be a duration string (e.g. '2s', '500ms'):"
                %w", err)
        }
        d.Duration = time.Duration(ns)
        return nil
    }

    if raw == "" {
        d.Duration = 0
        return nil
    }

    parsed, err := time.ParseDuration(raw)
    if err != nil {
        return fmt.Errorf("invalid delay %q: %w", raw, err)
    }
    d.Duration = parsed
    return nil
}

func (d *Duration) MarshalYAML() (interface{}, error) {
    return d.Duration.String(), nil
}

```

internal/scenario/parser.go

```

package scenario

import (
    "fmt"
    "net/http"
    "os"
    "slices"
    "strconv"
    "strings"
    "time"

    "gopkg.in/yaml.v3"
)

```

```

type Parser struct {
    scenario *Scenario
}

func NewParser() *Parser {
    return &Parser{}
}

func (p *Parser) ParseFile(filepath string) error {
    data, err := os.ReadFile(filepath)
    if err != nil {
        return fmt.Errorf("failed to read file: %w", err)
    }

    return p.ParseData(data)
}

func (p *Parser) ParseData(data []byte) error {
    var scenario Scenario
    if err := yaml.Unmarshal(data, &scenario); err != nil {
        return fmt.Errorf("failed to parse YAML: %w", err)
    }

    p.scenario = &scenario
    return nil
}

func (p *Parser) GetScenario() (*Scenario, error) {
    if p.scenario == nil {
        return nil, fmt.Errorf("no scenario loaded")
    }
    return p.scenario, nil
}

func (s *Scenario) FindStep(request string) *Step {
    for i := range s.Steps {
        if s.Steps[i].Request == request {
            return &s.Steps[i]
        }
    }
    return nil
}

const maxDelay = 10 * time.Minute

func (p *Parser) Validate() error {
    if p.scenario == nil {
        return fmt.Errorf("no scenario loaded")
    }

    if p.scenario.Name == "" {
        return fmt.Errorf("scenario.name is required")
    }

    if p.scenario.BaseURL == "" {
        return fmt.Errorf("scenario.base_url is required")
    }

    if p.scenario.VirtualUsers <= 0 {
        return fmt.Errorf("scenario.virtual_users must be greater than 0")
    }

    if p.scenario.Duration <= 0 {
        return fmt.Errorf("scenario.duration must be greater than 0")
    }
}

```

```

    if p.scenario.Duration > 31_556_952 {
        return fmt.Errorf("scenario.duration must be less than 1 year (31556952
seconds)")
    }

    if len(p.scenario.Steps) == 0 {
        return fmt.Errorf("scenario.steps: at least one step is required")
    }

    uniqueRequests := make(map[string]struct{})

    for i := range p.scenario.Steps {
        step := &p.scenario.Steps[i]

        if step.Request == "" {
            return fmt.Errorf("step[%d]: request field is required", i)
        }

        if _, exists := uniqueRequests[step.Request]; exists {
            return fmt.Errorf("step[%d]: duplicate request '%s'", i, step.Request)
        }
        uniqueRequests[step.Request] = struct{}{}

        httpMethod, _, err := parseRequest(step.Request)
        if err != nil {
            return fmt.Errorf("step[%d]: %w", i, err)
        }

        if (httpMethod == http.MethodGet || httpMethod == http.MethodHead) &&
            step.Body != nil {
            return fmt.Errorf("step[%d] (%s): GET and HEAD requests cannot have a
body",
                i, step.Request)
        }

        if step.Delay.Duration < 0 {
            return fmt.Errorf("step[%d] (%s): delay must be non-negative", i,
step.Request)
        }

        if step.Delay.Duration > maxDelay {
            return fmt.Errorf("step[%d] (%s): delay must not exceed %s", i,
step.Request, maxDelay)
        }

        for j := range step.NextSteps {
            nextStep := &step.NextSteps[j]

            hasReq := nextStep.Request != ""
            hasTag := nextStep.Tag != ""
            if hasReq == hasTag {
                return fmt.Errorf("step[%d], next_step[%d]: exactly one of `request`
or `tag` is required", i, j)
            }

            if hasReq {
                _, _, err := parseRequest(nextStep.Request)
                if err != nil {
                    return fmt.Errorf("step[%d], next_step[%d]: %w", i, j, err)
                }
                targetStep := p.scenario.FindStep(nextStep.Request)
                if targetStep == nil {
                    return fmt.Errorf("step[%d], next_step[%d]: target step '%s' not
found",
                        i, j, nextStep.Request)
                }
            } else {
                matched := p.scenario.StepsByTag(nextStep.Tag)

```

```

        if len(matched) == 0 {
            return fmt.Errorf("step[%d], next_step[%d]: no steps tagged '%s'",
                i, j, nextStep.Tag)
        }
        if nextStep.Pick != "" && nextStep.Pick != "random" && nextStep.Pick
!= "sequential" {
            return fmt.Errorf("step[%d], next_step[%d]: pick must be 'random'
or 'sequential' (got %q)",
                i, j, nextStep.Pick)
        }
    }

    for k, code := range nextStep.StatusCodes {
        if err := validateStatusCode(code); err != nil {
            return fmt.Errorf("step[%d], next_step[%d], status_code[%d]: %w",
                i, j, k, err)
        }
    }

    for mapSource, mapTarget := range nextStep.Map {
        if err := validateMapping(mapSource, mapTarget); err != nil {
            return fmt.Errorf("step[%d], next_step[%d]: invalid mapping '%s' -
> '%s': %w",
                i, j, mapSource, mapTarget, err)
        }
    }
}

return nil
}

func parseRequest(request string) (method string, path string, err error) {
    if request == "" {
        return "", "", fmt.Errorf("request cannot be empty")
    }

    parts := strings.SplitN(request, " ", 2)
    if len(parts) != 2 {
        return "", "", fmt.Errorf("invalid request format '%s', expected 'METHOD
/path'", request)
    }

    method = parts[0]
    path = parts[1]

    validMethods := []string{
        http.MethodGet,
        http.MethodPost,
        http.MethodPut,
        http.MethodPatch,
        http.MethodDelete,
        http.MethodHead,
    }

    if !slices.Contains(validMethods, method) {
        return "", "", fmt.Errorf("invalid HTTP method '%s', must be one of: %v",
            method, validMethods)
    }

    if !strings.HasPrefix(path, "/") {
        return "", "", fmt.Errorf("path must start with '/', got: %s", path)
    }

    return method, path, nil
}

func validateStatusCode(code string) error {

```

```

if code == "" {
    return fmt.Errorf("status code cannot be empty")
}

if len(code) == 3 && code[1:] == "xx" {
    firstChar := code[0]
    if firstChar < '1' || firstChar > '5' {
        return fmt.Errorf("wildcard must be 1xx-5xx, got: %s", code)
    }
    return nil
}

statusCode, err := strconv.Atoi(code)
if err != nil {
    return fmt.Errorf("invalid status code format '%s'", code)
}

if statusCode < 100 || statusCode >= 600 {
    return fmt.Errorf("status code must be 100-599, got: %d", statusCode)
}

return nil
}

func validateMapping(source, target string) error {
    validSources := []string{"response", "headers", "query", "path_params", "body",
"cookies", "variables"}
    validTargets := []string{"headers", "query", "path_params", "body", "cookies",
"variables"}

    sourceParts := strings.SplitN(source, ".", 2)
    if len(sourceParts) != 2 {
        return fmt.Errorf("invalid source format, expected 'source.field', got: %s",
source)
    }

    if !slices.Contains(validSources, sourceParts[0]) {
        return fmt.Errorf("invalid source '%s', must be one of: %v",
            sourceParts[0], validSources)
    }

    targetParts := strings.SplitN(target, ".", 2)
    if len(targetParts) != 2 {
        return fmt.Errorf("invalid target format, expected 'target.field', got: %s",
target)
    }

    if !slices.Contains(validTargets, targetParts[0]) {
        return fmt.Errorf("invalid target '%s', must be one of: %v",
            targetParts[0], validTargets)
    }

    return nil
}

```

internal/executor/executor.go

```
package executor
```

```
import (
    "bytes"
    "context"
    "fmt"
    "io"
    "net/http"
    "net/http/cookiejar"
    "time"

```

```

)

type HTTPClient interface {
    Do(req *http.Request) (*http.Response, error)
}

type Request struct {
    Method  string
    URL     string
    Headers map[string]string
    Body    []byte
    Timeout time.Duration
}

type Response struct {
    StatusCode int
    Status     string
    Headers    map[string][]string
    Body       []byte
    Duration   time.Duration
}

type Executor struct {
    client HTTPClient
    jar     http.CookieJar
}

func New() (*Executor, error) {
    jar, err := cookiejar.New(nil)
    if err != nil {
        return nil, fmt.Errorf("failed to create cookie jar: %w", err)
    }

    client := &http.Client{
        Jar:     jar,
        Timeout: 30 * time.Second,
    }

    return &Executor{
        client: client,
        jar:    jar,
    }, nil
}

func NewWithClient(client HTTPClient) *Executor {
    return &Executor{
        client: client,
    }
}

func (e *Executor) Execute(ctx context.Context, req *Request) (*Response, error) {
    if req == nil {
        return nil, fmt.Errorf("request cannot be nil")
    }

    if req.URL == "" {
        return nil, fmt.Errorf("URL cannot be empty")
    }

    if req.Method == "" {
        req.Method = http.MethodGet
    }

    var bodyReader io.Reader
    if req.Body != nil {
        bodyReader = bytes.NewReader(req.Body)
    }

```

```

httpReq, err := http.NewRequestWithContext(ctx, req.Method, req.URL, bodyReader)
if err != nil {
    return nil, fmt.Errorf("failed to create request: %w", err)
}

for key, value := range req.Headers {
    httpReq.Header.Set(key, value)
}

if req.Timeout > 0 {
    var cancel context.CancelFunc
    ctx, cancel = context.WithTimeout(ctx, req.Timeout)
    defer cancel()
    httpReq = httpReq.WithContext(ctx)
}

start := time.Now()
httpResp, err := e.client.Do(httpReq)
duration := time.Since(start)

if err != nil {
    return nil, fmt.Errorf("request failed: %w", err)
}
defer httpResp.Body.Close()

respBody, err := io.ReadAll(httpResp.Body)
if err != nil {
    return nil, fmt.Errorf("failed to read response body: %w", err)
}

response := &Response{
    StatusCode: httpResp.StatusCode,
    Status:     httpResp.Status,
    Headers:    httpResp.Header,
    Body:       respBody,
    Duration:   duration,
}

return response, nil
}

func (e *Executor) GET(ctx context.Context, url string, headers map[string]string)
(*Response, error) {
    req := &Request{
        Method: http.MethodGet,
        URL:    url,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) POST(ctx context.Context, url string, body []byte, headers
map[string]string) (*Response, error) {
    req := &Request{
        Method: http.MethodPost,
        URL:    url,
        Body:   body,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) PUT(ctx context.Context, url string, body []byte, headers
map[string]string) (*Response, error) {
    req := &Request{
        Method: http.MethodPut,
        URL:    url,
        Body:   body,
    }

```

```

        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) PATCH(ctx context.Context, url string, body []byte, headers
map[string]string) (*Response, error) {
    req := &Request{
        Method: http.MethodPatch,
        URL:     url,
        Body:    body,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) DELETE(ctx context.Context, url string, headers map[string]string)
(*Response, error) {
    req := &Request{
        Method: http.MethodDelete,
        URL:     url,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) HEAD(ctx context.Context, url string, headers map[string]string)
(*Response, error) {
    req := &Request{
        Method: http.MethodHead,
        URL:     url,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) OPTIONS(ctx context.Context, url string, headers map[string]string)
(*Response, error) {
    req := &Request{
        Method: http.MethodOptions,
        URL:     url,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) TRACE(ctx context.Context, url string, headers map[string]string)
(*Response, error) {
    req := &Request{
        Method: http.MethodTrace,
        URL:     url,
        Headers: headers,
    }
    return e.Execute(ctx, req)
}

func (e *Executor) GetCookieJar() http.CookieJar {
    return e.jar
}

```

internal/extractor/extractor.go

```

package extractor

import (
    "fmt"

    "github.com/tidwall/gjson"

```

```

)

type Extractor struct{}

func New() *Extractor {
    return &Extractor{}
}

func (e *Extractor) Extract(jsonData []byte, path string) (any, error) {
    if len(jsonData) == 0 {
        return nil, fmt.Errorf("json data cannot be empty")
    }

    if path == "" {
        return nil, fmt.Errorf("path cannot be empty")
    }

    result := gjson.GetBytes(jsonData, path)

    if !result.Exists() {
        return nil, fmt.Errorf("path '%s' not found in JSON", path)
    }

    return result.Value(), nil
}

func (e *Extractor) Exists(jsonData []byte, path string) bool {
    if len(jsonData) == 0 || path == "" {
        return false
    }

    result := gjson.GetBytes(jsonData, path)
    return result.Exists()
}

func (e *Extractor) ExtractWithDefault(jsonData []byte, path string, defaultValue any)
any {
    value, err := e.Extract(jsonData, path)
    if err != nil {
        return defaultValue
    }
    return value
}

```

internal/runner/runner.go

```

package runner

import (
    "context"
    "fmt"
    "os"
    "os/signal"
    "sync"
    "syscall"
    "time"

    "loadforge-agent/internal/scenario"
)

type Runner struct {
    scenario *scenario.Scenario
    stats    *Stats
}

func New(sc *scenario.Scenario) *Runner {
    return &Runner{scenario: sc, stats: NewStats()}
}

```

```

}

func (r *Runner) Stats() *Stats { return r.stats }

func (r *Runner) Run(ctx context.Context) error {
    sc := r.scenario
    if sc == nil {
        return fmt.Errorf("scenario is nil")
    }

    duration := time.Duration(sc.Duration) * time.Second
    numVUs := int(sc.VirtualUsers)

    rampDuration := duration / 10
    if rampDuration < time.Millisecond {
        rampDuration = 0
    }
    var rampInterval time.Duration
    if numVUs > 1 && rampDuration > 0 {
        rampInterval = rampDuration / time.Duration(numVUs)
    }

    ctx, cancel := context.WithTimeout(ctx, duration)
    defer cancel()

    sigCh := make(chan os.Signal, 1)
    signal.Notify(sigCh, syscall.SIGINT, syscall.SIGTERM)
    go func() {
        select {
        case <-sigCh:
            cancel()
        case <-ctx.Done():
        }
    }()

    var wg sync.WaitGroup

    for i := range numVUs {
        if i > 0 && rampInterval > 0 {
            select {
            case <-ctx.Done():
                break
            case <-time.After(rampInterval):
            }
        }

        if ctx.Err() != nil {
            break
        }

        vu, err := newVirtualUser(uint64(i+1), sc, r.stats)
        if err != nil {
            return fmt.Errorf("failed to create vu %d: %w", i+1, err)
        }

        wg.Add(1)
        go func(vu *VirtualUser) {
            defer wg.Done()
            runVU(ctx, vu)
        }(vu)
    }

    wg.Wait()
    signal.Stop(sigCh)
    return nil
}

func runVU(ctx context.Context, vu *VirtualUser) {

```

```

    for {
        if ctx.Err() != nil {
            return
        }
        _ = vu.runScenario(ctx)
    }
}

```

internal/openapi/parser.go

```

package openapi

import (
    "context"
    "fmt"
    "os"
    "slices"

    "github.com/getkin/kin-openapi/openapi3"
)

type Endpoint struct {
    Path      string
    Method    string
    Tags      []string
    Responses any

    OperationID string
    Summary      string
    Description  string
}

type Parser struct {
    doc *openapi3.T
}

func New() *Parser {
    return &Parser{}
}

func (p *Parser) ParseFile(filepath string) error {
    data, err := os.ReadFile(filepath)
    if err != nil {
        return fmt.Errorf("failed to read file: %w", err)
    }

    loader := openapi3.NewLoader()
    loader.IsExternalRefsAllowed = true

    doc, err := loader.LoadFromData(data)
    if err != nil {
        return fmt.Errorf("failed to parse OpenAPI spec: %w", err)
    }

    ctx := context.Background()
    if err := doc.Validate(ctx); err != nil {
        return fmt.Errorf("invalid OpenAPI spec: %w", err)
    }

    p.doc = doc
    return nil
}

func (p *Parser) ParseData(data []byte) error {
    loader := openapi3.NewLoader()
    loader.IsExternalRefsAllowed = true

```

```

doc, err := loader.LoadFromData(data)
if err != nil {
    return fmt.Errorf("failed to parse OpenAPI spec: %w", err)
}

ctx := context.Background()
if err := doc.Validate(ctx); err != nil {
    return fmt.Errorf("invalid OpenAPI spec: %w", err)
}

p.doc = doc
return nil
}

func (p *Parser) GetEndpoints() ([]Endpoint, error) {
    if p.doc == nil {
        return nil, fmt.Errorf("no document loaded")
    }

    if p.doc.Paths == nil {
        return []Endpoint{}, nil
    }

    var endpoints []Endpoint

    for path, pathItem := range p.doc.Paths.Map() {
        if pathItem == nil {
            continue
        }

        operations := map[string]*openapi3.Operation{
            "GET":    pathItem.Get,
            "POST":   pathItem.Post,
            "PUT":    pathItem.Put,
            "PATCH": pathItem.Patch,
            "DELETE": pathItem.Delete,
            "HEAD":   pathItem.Head,
            "OPTIONS": pathItem.Options,
            "TRACE":  pathItem.Trace,
        }

        for method, operation := range operations {
            if operation == nil {
                continue
            }

            endpoint := Endpoint{
                Path:        path,
                Method:      method,
                OperationID: operation.OperationID,
                Summary:     operation.Summary,
                Description: operation.Description,
                Tags:        operation.Tags,
            }

            endpoints = append(endpoints, endpoint)
        }
    }

    return endpoints, nil
}

func (p *Parser) GetEndpointsByTag(tag string) ([]Endpoint, error) {
    allEndpoints, err := p.GetEndpoints()
    if err != nil {
        return nil, err
    }

```

```
var filtered []Endpoint
for _, endpoint := range allEndpoints {
    if slices.Contains(endpoint.Tags, tag) {
        filtered = append(filtered, endpoint)
    }
}

return filtered, nil
}
```