УДК 004.415.5, 004.42, 681.513

РАЗРАБОТКА ФРЕЙМВОРКА ДЛЯ АДАПТИВНОГО УПРАВЛЕНИЯ НАГРУЗОЧНЫМ ТЕСТИРОВАНИЕМ

Смирнов Вадим Николаевич[1], Ткач Оксана Владимировна[2]

[1]Студент;
Государственный университет «Дубна»;
141980, Россия, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: svn.22@uni-dubna.ru.

[2]Старший преподаватель;
Государственный университет «Дубна»;
141980, Россия, Московская обл., г. Дубна, ул. Университетская, 19;
e-mail: kseny-rdm@uni-dubna.ru.

В данной работе представлена разработка фреймворка для адаптивного нагрузочного тестирования, реализующего замкнутый цикл управления: сбор метрик – анализ состояния – принятие решения – корректировка нагрузки. В отличие от существующих инструментов, система не требует заранее заданного сценария и самостоятельно определяет предел производительности тестируемого сервиса в реальном времени. Фреймворк построен на основе паттернов Adapter и Strategy, включает пять стратегий управления нагрузкой и TUI-интерфейс.

Ключевые слова: адаптивное нагрузочное тестирование, замкнутый цикл управления, оркестрация тестирования, стратегии управления нагрузкой, TUI-интерфейс, Locust.

Для цитирования:

Смирнов В. Н., Ткач О. В. Разработка фреймворка для адаптивного управления нагрузочным тестированием // Системный анализ в науке и образовании. 2026. № 2. С. 15-22. EDN: OTKTIF. URL: https://sanse.ru/index.php/sanse/article/view/714.

DEVELOPMENT OF A FRAMEWORK FOR ADAPTIVE LOAD TESTING CONTROL

Smirnov Vadim N.[1], Tkach Oxana V.[2]

[1]Student;
Dubna State University;
19 UniversitetskayaStr., Dubna, Moscow region, 141980, Russia;
e-mail: svn.22@uni-dubna.ru

[2]Senior teacher;
Dubna State University;
19 Universitetskaya Str., Dubna, Moscow region, 141980, Russia;
e-mail:kseny-rdm@uni-dubna.ru

This paper presents the development of a framework for adaptive load testing that implements a closed-loop control cycle: metrics collection – state analysis – decision making – load adjustment. Unlike existing tools, the system does not require a predefined test scenario and autonomously determines the performance limit of the service under test in real time. The framework is built on the Adapter and Strategy patterns, includes five load management strategies, and provides a TUI (Text User Interface).

Keywords: adaptive load testing, closed-loop control, test orchestration, load management strategies, TUI, Locust.

For citation:
Smirnov V. N., Tkach O. V. Development of a Framework for Adaptive Load Testing Control. *System analysis in science and education*, 2026;(2):15-22 (in Russ). EDN: OTKTIF. Available from: https://sanse.ru/index.php/sanse/article/view/714.

Введение

Нагрузочное тестирование является обязательным этапом проверки производительности программных систем. Традиционный подход предполагает, что инженер заранее определяет профиль нагрузки, запускает тест по фиксированному сценарию и анализирует результаты после его завершения [1]. Этот подход имеет существенное ограничение: реальные возможности системы неизвестны до начала теста. В результате нагрузка либо выбирается с запасом (и предел производительности не достигается), либо система уходит в отказ раньше, чем инженер успевает среагировать.

Проблема усугубляется в условиях микросервисных архитектур, где поведение системы под нагрузкой сложно предсказать заранее. Существующие инструменты – генераторы нагрузки (*Locust[2]*, *JMeter[3]*, *K6[4]*), платформы запуска (*Taurus[5]*, *Testkube[6]*) и коммерческие решения – не решают задачу адаптивного управления в реальном времени: одни работают строго по заданному сценарию, другие анализируют метрики только после завершения теста.

Целью данной работы является разработка фреймворка для аналитической регулировки нагрузки посредством анализа метрик, получаемых от генераторов нагрузки. Система функционирует как интеллектуальная надстройка (оркестратор) над существующими генераторами, образуя замкнутый цикл управления. Ожидаемый результат — функциональный прототип, обеспечивающий адаптивное управление нагрузкой, раннее обнаружение деградации и интеллектуальную аналитику в реальном времени.

Актуальность работы обусловлена переходом от реактивного управления качеством к проактивному: в современных *DevOps*- и микросервисных средах ручное управление нагрузочным тестированием становится узким местом, а отказ системы в пиковые нагрузки может привести к значительным финансовым и репутационным потерям. Разрабатываемый фреймворк позволяет автоматически находить предел производительности без участия инженера, что критически важно для непрерывной интеграции и регулярного тестирования производительности.

1. Анализ существующих решений

Существующие инструменты нагрузочного тестирования можно разделить на три категории [7].

Генераторы нагрузки (*Locust[2]*, *JMeter[3]*, *K6[4]*) обеспечивают гибкое описание сценариев на языках программирования и базовую агрегацию метрик. Однако они работают по заранее определённому сценарию, не имеют механизма принятия решений на основе метрик в реальном времени и требуют ручной настройки параметров теста.

Платформы запуска (*Taurus[5]*, *Testkube[6]*) унифицируют запуск различных генераторов и интегрируются с *CI/CD*. Но оценка результатов производится только после завершения теста, отсутствует динамическое управление профилем нагрузки во время выполнения.

Коммерческие платформы (*Tricentis NeoLoad*, *PFLB Platform*) предлагают продвинутую аналитику и автоматическое обнаружение аномалий, однако ориентированы на постфактум-анализ, а не на управление в реальном времени. Кроме того, они характеризуются высокой стоимостью и сложностью интеграции с *open-source* инструментами.

В таблице 1 представлено сравнение решений по ключевым критериям.

Табл. 1. Сравнительный анализ существующих решений

Критерий	Генераторы нагрузки	Платформы запуска	Коммерческие платформы	Разрабатываемый продукт
Адаптивность реальном времени	0	0	1	1

Автоопределение пределов	0	0	0	1
Анализ метрик	0	0	0.5	1
Автономность	0	1	1	1
Интеграция с генераторами	–	1	1	1
Открытость	1	1	0	1

Вывод: на рынке отсутствуют инструменты, способные автоматически адаптировать нагрузку в реальном времени и самостоятельно определять пределы производительности системы. Разрабатываемый фреймворк занимает эту нишу, выступая в роли интеллектуального управляющего элемента над существующими генераторами нагрузки.

2. Требования к системе

На основе анализа потребностей целевой аудитории (инженеры автоматизации тестирования и технические лидеры) сформулированы следующие **функциональные требования**:

1. Командный запуск и режимы работы – система функционирует как *CLI*-приложение, поддерживает выполнение теста, валидацию конфигурации и просмотр результатов.
2. Конфигурационное управление через *YAML* – все входные параметры задаются через *YAML*-файл.
3. Управление генератором нагрузки – автоматический запуск и остановка генератора без участия пользователя.
4. Динамическое изменение нагрузки – изменение количества виртуальных пользователей и скорости их создания во время выполнения теста.
5. Периодический сбор метрик – получение метрик производительности и надёжности с заданной периодичностью.
6. Нормализация и агрегация метрик – приведение метрик к унифицированному формату.
7. Хранение истории выполнения – сохранение временной истории метрик, фаз нагрузки и управляющих решений.
8. Плагинная модель стратегий – поддержка подключаемых стратегий управления, не зависящих от реализации генератора.
9. Анализ текущего состояния системы – определение состояния на основе метрик и их динамики.
10. Принятие управляющих решений – формирование решений: увеличение, уменьшение, удержание или остановка нагрузки.
11. Аварийное завершение теста – немедленная остановка при достижении критических условий.

Нефункциональные требования: версия *Python* не ниже 3.10, пользовательский интерфейс реализован как *TUI* (*Text User Interface*), архитектура позволяет добавлять новые экранные формы и стратегии без изменения существующих компонентов.

Требования к пользовательскому интерфейсу включают три основных экрана: экран конфигурации (ввод параметров), экран выполнения теста (мониторинг в реальном времени) и экран графиков (визуализация метрик). Навигация осуществляется горячими клавишами *F1 – F3*.

3. Архитектура системы

Разработанная система построена на модульной архитектуре с использованием двух ключевых паттернов проектирования.

Паттерн *Adapter* отвечает за взаимодействие с конкретным генератором нагрузки и преобразование полученных метрик в унифицированный формат *RawMetrics*. Это позволяет

фреймворку оставаться независимым от конкретного инструмента генерации трафика – для поддержки нового генератора достаточно реализовать соответствующий адаптер.

Паттерн *Strategy* инкапсулирует логику управления нагрузкой. Стратегия получает нормализованные метрики и возвращает управляющее решение (продолжение, остановка, удержание нагрузки) а также целевое количество пользователей. Стратегия не имеет информации об адаптере, что обеспечивает её переиспользование.

Оркестратор является центральным компонентом, реализующим замкнутый цикл управления (рисунок 1):

1. Сбор метрик через адаптер
2. Передача метрик в стратегию
3. Получение управляющего решения
4. Корректировка нагрузки

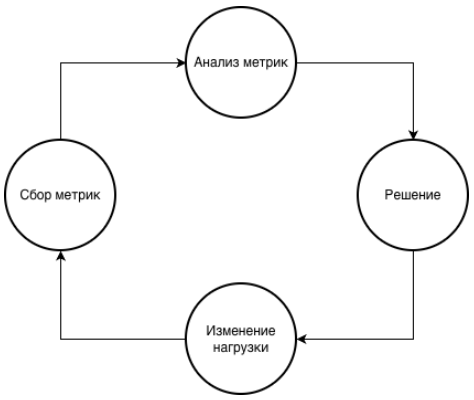

Рис. 1. Процесс работы системы

Оркестратор проходит три фазы:

- *INIT* – запуск генератора, ожидание готовности, установка начальной нагрузки и стабилизационная пауза.
- *RUNNING* – циклическое выполнение шагов сбора метрик, принятия решений и изменения нагрузки.
- *FINISHED* – остановка генератора и формирование результата теста.

Такая архитектура обеспечивает расширяемость: добавление новых стратегий или адаптеров не требует изменений в ядре системы.

4. Реализация

4.1 Реализация адаптера *Locust*

Адаптер для *Locust[2]* взаимодействует с генератором через встроенный *REST API*. Запуск теста и изменение нагрузки выполняются через *POST* /swarm с параметрами *user_count* и *spawn_rate*. Текущие метрики считываются через *GET /stats/requests*. Из ответа извлекается агрегированная строка по всем эндпоинтам: *RPS*, среднее время ответа, перцентили *P50/P95/P99*, число запросов и ошибок. Все данные упаковываются в датакласс *RawMetrics*.

4.2 Стратегии управления нагрузкой

Реализовано пять стратегий, решающих различные задачи нагрузочного тестирования. Все стратегии реализуют интерфейс *IStrategy*. Оркестратор вызывает только четыре метода и не зависит от конкретной реализации.

DegradationSearch – поиск точки деградации системы, когда рост нагрузки приводит к ухудшению времени ответа. Алгоритм накапливает историю измерений и вычисляет базовое значение

P95 (*baseline*) за предпоследние 5 – 10 точек, затем сравнивает с последними тремя значениями. Если все три последних *P95* превышают *baseline* в 1,5 раза, система фиксирует деградацию. Рост нагрузки может быть линейным или экспоненциальным. Обоснование выбора перцентилей основано на теории массового обслуживания[1]: хвосты распределения времени ответа (*P95*, *P99*) реагируют на рост очередей первыми, что позволяет обнаружить деградацию раньше, чем это станет очевидно по средним значениям.

BreakPoint – определение предела полного отказа. Стратегия агрессивно повышает нагрузку и проверяет четыре условия: превышение порога ошибок (по умолчанию 10%), падение *RPS* до нуля при ненулевом количестве пользователей, экстремальная задержка на *P99* (>10 секунд), снижение пропускной способности более чем вдвое относительно предыдущего измерения.

SLAValidation – проверка соответствия заданным *SLA*. Каждая ступень нагрузки проходит две фазы: стабилизация (игнорирование переходных процессов) и наблюдение (накопление метрик). По завершении фазы наблюдения вычисляются средние значения *P99* и *error_rate* и сравниваются с порогами из конфигурации.

Canary – контроль стабильности при минимальной фиксированной нагрузке. Система наблюдается в течение заданного времени; при нарушении порогов *P99* или *error_rate* тест немедленно останавливается.

Spike – проверка поведения системы при резких всплесках нагрузки. Тест проходит три фазы: прогрев на стабильной нагрузке, пиковое значение, восстановление до минимального уровня. Оценивается как поведение под пиком, так и способность системы восстановиться после снижения нагрузки.

4.3 Оркестратор

Оркестратор управляет жизненным циклом теста, работая исключительно через интерфейсы *IAdapter* и *IStrategy*. На каждой итерации он запрашивает метрики, передаёт их в стратегию и обрабатывает решение. При решении *CONTINUE* запрашивается новое количество пользователей, при *HOLD* нагрузка не меняется, при *STOP* тест завершается.

Помимо решений стратегии, оркестратор самостоятельно проверяет глобальные критические условия: превышение максимального числа пользователей, высокий процент ошибок, полную неработоспособность системы. Любое необработанное исключение в цикле приводит к корректной остановке генератора.

4.4 TUI-интерфейс

Пользовательский интерфейс реализован с использованием библиотеки *Textual[8]*. Он включает три экрана:

- Экран конфигурации (*F1*) – ввод параметров теста: путь к *YAML*-конфигу, адаптер, тест-файл, хост, порт, стратегия. При выборе стратегии форма параметров перестраивается динамически.
- Экран выполнения (*F2*) – отображение текущего состояния: активные пользователи, *RPS*, *P95*, процент ошибок, а также прокручиваемый лог событий с временными метками.
- Экран графиков (*F3*) – три вкладки: совмещённый график *RPS* и пользователей, графики перцентилей времени ответа, графики показателей ошибок.

4.5 Конфигурирование через YAML

Все параметры теста задаются в *YAML*-файле, разделённом на три секции: *adapter*, *orchestrator* и *strategy*.

Секция *adapter* описывает подключение к генератору нагрузки: хост, порт и путь к тест-файлу. Секция *orchestrator* задаёт общие параметры цикла: начальное число пользователей, шаг и интервал опроса метрик. Секция *strategy* содержит название стратегии и её параметры – набор полей зависит от выбранной стратегии и валидируется при загрузке.

Валидация входных параметров выполняется на этапе загрузки – ошибки конфигурации выводятся до старта теста.

5. Технологии реализации

- *Python* – основной язык разработки; выбран благодаря обширной экосистеме библиотек для работы с API, хорошей поддержке асинхронности и удобству написания модульных тестов.
- *Textual[8]* – библиотека для построения *TUI*-приложений; обеспечивает реактивный интерфейс с поддержкой виджетов, графиков и горячих клавиш.
- *PyYAML* – парсинг *YAML*-конфигурационных файлов.
- *asyncio* – асинхронный сбор метрик и управление циклом оркестратора.
- *pytest* – фреймворк для модульного тестирования.

6. Тестирование

Для проверки корректности работы компонентов использовалось модульное автоматизированное тестирование с фреймворком *pytest*. Каждый тест изолированно проверяет поведение отдельного компонента (стратегии тестируются без реального генератора). Набор тестовых сценариев покрывает функциональные требования, описанные в разделе требований.

Разработаны и реализованы следующие тест-кейсы:

- Конфигурация: 8 тест-кейсов (загрузка корректного/некорректного *YAML*, проверка обязательных полей, конвертация)
- Адаптер: 5 тест-кейсов (проверка готовности, преобразование метрик, корректная остановка)
- Стратегии: 30 тест-кейсов (по 5 – 8 на каждую стратегию, покрывающие все условия принятия решений)
- Оркестратор: 14 тест-кейсов (жизненный цикл, обработка решений, аварийное завершение, хранение истории)

Все тесты успешно пройдены (рисунок 2). Матрица трассировки требований подтверждает покрытие всех функциональных требований разработанными тест-кейсами.

Тестирование пользовательского интерфейса проводилось в формате *UI/UX*-оценки. Навигация между экранами через *F1/F2/F3* работает ожидаемо, метрики на экране выполнения обновляются в реальном времени и читаются без затруднений. Критических замечаний по интерфейсу не выявлено.

Практическая значимость подтверждена тестированием на реальном проекте – фреймворк успешно использован для поиска узких мест производительности.

Заключение

В статье представлен фреймворк для адаптивного нагрузочного тестирования, реализующий замкнутый цикл управления «сбор метрик → анализ → решение → корректировка нагрузки». Ключевой результат работы — переход от статических, заранее заданных сценариев к динамическому управлению интенсивностью тестирования на основе реального поведения системы.

В ходе разработки достигнуто следующее:

- Построена модульная архитектура с паттернами *Adapter* и *Strategy*, обеспечивающая независимую подключаемость генераторов нагрузки (на примере *Locust[2]*) и стратегий управления.
- Реализованы пять стратегий, покрывающих основные сценарии нагрузочного тестирования: поиск точки деградации, определение предела отказа, проверка соответствия *SLA*, стабильность при минимальной нагрузке и реакция на резкие всплески.
- Создан *TUI*-интерфейс[8] для мониторинга в реальном времени и *YAML*-конфигуратор, позволяющий запускать тесты без изменения кода.
- Корректность работы подтверждена 60 модульными тестами и апробацией на реальном проекте.

Матрица трассировки – Конфигурация

Требование	1. Загрузка корректного YAML-конфига	2. Загрузка несуществующего файла	3. Без блока adapter	4. Без блока strategy	5. Несуществующий тестовый файл	6. Без блока orchestrator	7. Конвертация в словарь и обратно	8. Конвертация в YAML и обратно
1. Командный запуск	+	+						
2. Конфигурация через YAML	+	+	+	+	+	+	+	+

Матрица трассировки – Адаптер

Требование	1. True при готовности	2. False при не готовности	3. Преобразование метрик в RawMetrics	4. fail ratio в проценты	5. Завершение процесса при остановке
3. Управление генератором	+	+			+
4. Динамическое изменение нагрузки					
5. Периодический сбор метрик	+	+	+		
6. Нормализация метрик			+	+	

Матрица трассировки – Стратегии

Требование	1. CONTINUE при недостатке данных (DS)	2. CONTINUE при стабильных метриках (DS)	3. STOP при росте p95 (DS)	4. STOP при стеке ошибок (DS)	5. Начальный пользователи при 0 (DS)	6. Линейный рост (DS)	7. Экспоненциальный рост (DS)	8. reset очищает историю (DS)	9. STOP при error_rate (BP)	10. STOP при RPS=0 (BP)	11. STOP при p99 > 10с (BP)	12. STOP при падении RPS 50% (BP)	13. Начальные пользователи при 0 (BP)	14. reset сбрасывает метрики (BP)	15. CONTINUE в стабилизации (SLA)	16. STOP при нарушении p99 (SLA)	17. STOP при нарушении ошибок (SLA)	18. CONTINUE при соблюдении SLA	19. ограничение max_users (SLA)	20. Пользователи BASELINE (Spike)	21. Переход BASELINE → SPIKE (Spike)	22. Пользователи SPIKE (Spike)	23. Переход SPIKE → RECOVERY (Spike)	24. STOP после RECOVERY (Spike)	25. reset → BASELINE (Spike)	26. HOLD без нарушений (Canary)	27. STOP при ошибках (Canary)	28. STOP при p99 (Canary)	29. STOP по времени (Canary)	30. Фиксированные пользователи (Canary)
8. Плагинная модель стратегий	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
9. Анализ состояния системы	+	+	+	+											+	+	+				+	+		+	+		+			

Матрица трассировки – Оркестратор

Требование	1. Возврат результата после теста	2. Результат при ошибке адаптера	3. Причина ERROR при исключении	4. STOP от стратегии	5. STOP при критических ошибках	6. STOP при RPS=0	7. STOP при превышении max_users	8. STOP по таймауту	9. Нет изменений при HOLD	10. get_next_users при CONTINUE	11. Нет изменений до wait_time	12. Причина остановки в результате	13. Причина MANUAL при stop()	14. История = кол-во метрик
4. Динамическое изменение нагрузки									+	+	+			
7. Хранение истории														+
10. Принятие решений				+					+	+	+			
11. Аварийное завершение		+	+		+	+								

Рис. 2. Матрица трассировки требований

Основное отличие от существующих инструментов — способность автономно, в реальном времени, определять предел производительности системы и останавливать тест при первых признаках деградации, что сокращает время анализа и исключает необходимость постоянного ручного контроля. Дальнейшее развитие фреймворка может включать поддержку дополнительных генераторов нагрузки (*JMeter*, *k6*), распределённый режим и интеграцию с системами мониторинга.

Список источников

1. Рагозин А. Теория и практика нагрузочного тестирования // Heisenbug 2024 Spring : Москва, 22 – 23 апреля 2024 г.
2. Locust Documentation. – Carl Byström, Jonatan Heyman, Lars Holmberg, 2009-2025. – URL: https://docs.locust.io/ (дата обращения: 15.11.2025).
3. Apache Jmeter: [documentation]. – Apache Software Foundation, 1999–2024. – URL: https://jmeter.apache.org/ (дата обращения: 25.11.2025).
4. Load testing for engineering teams | Grafana k6. – Grafana Labs, 2024. – URL: https://k6.io/docs/ (дата обращения: 25.11.2025).
5. User Manual: [Taurus Tool Documentation]. – BlazeMeter Inc., 2014-2025. – URL: https://gettaurus.org/docs/ (дата обращения: 27.11.2025).
6. Welcome to the Testkube Documentation! | Testkube Documentation. –Kubeshop, 2025. – URL: https://docs.testkube.io/ (дата обращения: 27.11.2025).
7. Филонов Н. Правильный инструмент для аналитики нагрузочного тестирования // ХАБР: [блог-платформа]. – Habr, 2006–2026. – URL: https://habr.com/ru/articles/833598/. – Опубл. sound_right 13.08.2024.
8. Textual: [documentation] // Textualize. – Textualize, Inc., [2026]. – URL: https://textual.textualize.io/ (дата обращения: 15.02.2026).