ПОЛТАВСЬКИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТОРГІВЛІ
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____Олена ОЛЬХОВСЬКА
(підпис)
«_____»___202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

ІНСТРУМЕНТ ДЛЯ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ ВЕБ- ДОДАТКІВ У РОЗПОДІЛЕНИХ СЕРЕДОВИЩАХ

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня магістра

Виконавець роботи Борисенко Ілля Володимирович
_____«____» _____202_ р.
(підпис)

Науковий керівник доцент, к.ф.-м.н. Черненко О. О.
_____«____»_____202_ р.
(підпис)

Рецензент

ПОЛТАВА 2025

РЕФЕРАТ

Записка: 73 с., 11 рис., 2 таблиці, 1 додаток, 13 джерел.

ТЕСТУВАННЯ ПРОДУКТИВНОСТІ, NODE.JS, DISTRIBUTED AGENTS

Об’єктом розробки є інструмент для автоматизації тестування продуктивності веб-додатків у розподілених середовищах.

Предметом розробки є програмна реалізація системи, що забезпечує створення сценаріїв навантаження, запуск тестів на кількох вузлах та аналіз результатів у зручному інтерфейсі.

Метою роботи є створення інструменту, який дозволяє автоматизувати процес навантажувального тестування, спростити налаштування сценаріїв і забезпечити інтерактивний аналіз метрик продуктивності у реальному часі.

Результатом роботи стало розроблення веб-системи PerfBench, яка складається з Control Plane (бекенд), веб-інтерфейсу та розподілених агентів. Інструмент дозволяє гнучко створювати сценарії тестування через форму або JSON-специфікацію, застосовувати готові шаблони навантаження (steady, ramp, spike, stress, flow-сценарії, traffic mix) та запускати їх на декількох агентських вузлах.

Однією з ключових особливостей є розподілене виконання тестів: агенти підключаються до Control Plane, отримують завдання та виконують навантаження, повертаючи bulk-метрики. Це дає можливість масштабувати тести й моделювати роботу сотень або тисяч одночасних користувачів.

У системі реалізовано збір та візуалізацію метрик: p50, p95, p99, throughput, error rate. Користувач бачить графіки затримок, пропускної здатності та частки помилок у часі, а також гістограму запитів. Передбачено перевірку SLA-порогів, що автоматично сигналізує про порушення умов (наприклад, p95 > 700 мс або error rate > 2%). Результати можна експортувати у формат CSV для подальшого аналізу.

Інтерфейс PerfBench містить розділи:

- Сценарії -створення та збереження власних конфігурацій;
- Запуски -контроль виконання тестів, зупинка та перегляд деталей;
- Шаблони -набір готових профілів навантаження;

- Посібник -інтегрована довідка та швидкий старт у 5 кроків.

Для перевірки роботи інструменту реалізовано demo-target, що імітує веб-сервіс із параметрами delay, jitter, fail. Він дозволяє без зовнішніх ресурсів перевіряти поведінку системи під різними типами навантаження.

PerfBench було протестовано у локальному та розподіленому середовищах. Робота підтверджена як стабільна й ефективна: система коректно обробляє десятки тисяч запитів, а інтерфейс залишається інтуїтивним та зручним для користувачів.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕННЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
SLA (Service Level Agreement)	Угода про рівень надання послуг, що визначає очікувані показники продуктивності (затримка, помилки тощо).
p95	95-й перцентиль часу відгуку -показник, що відображає, за який час виконується 95% запитів.
Throughput	Пропускна здатність системи, кількість успішних запитів за секунду (RPS).
Error rate	Частка запитів, що завершилися з помилкою (у відсотках).
RPS (Requests per second)	Кількість запитів на секунду.
UI (User Interface)	Інтерфейс користувача.
API (Application Programming Interface)	Програмний інтерфейс для взаємодії між компонентами системи.
JSON (JavaScript Object Notation)	Формат обміну даними, що використовується у сценаріях PerfBench.
Agent	Легкий процес-виконавець, який отримує завдання з Control Plane та генерує навантаження.
Control Plane	Центральний модуль, що керує агентами, сценаріями та обробляє результати тестів.
Load Testing	Метод тестування, що вимірює продуктивність системи під навантаженням.
Stress Testing	Тестування «на межі», що показує, як система поводиться при екстремальному навантаженні.

Spike Profile	Профіль навантаження з різким стрибком кількості запитів.
Ramp Profile	Профіль поступового зростання навантаження.
Steady Profile	Профіль сталого навантаження без змін у часі.

ВСТУП

У світі розвиток веб-технологій та зростання кількості онлайн-сервісів вимагають особливої уваги до продуктивності програмних систем. Користувачі очікують стабільного доступу, швидкого відгуку та надійності роботи веб-додатків незалежно від навантаження. Саме тому тестування продуктивності стало невід'ємною складовою процесу розробки та впровадження програмних продуктів.

Тестування продуктивності дозволяє виявляти «вузькі місця» системи, визначати її стійкість до стресових навантажень, прогнозувати поведінку при зростанні кількості користувачів. Проте використання існуючих інструментів часто супроводжується складною конфігурацією, потребою в додатковій інфраструктурі або відсутністю зручної аналітики. Це створює труднощі для команд розробників і тестувальників, які прагнуть швидко отримати достовірні результати.

Розробка інструменту PerfBench спрямована на спрощення процесу автоматизації навантажувального тестування. Система поєднує створення сценаріїв, запуск тестів, збір та візуалізацію метрик у єдиному середовищі. Додатковою перевагою є підтримка розподілених запусків, що дозволяє моделювати реальні умови роботи веб-додатків з великою кількістю користувачів.

Актуальність дослідження полягає у необхідності створення зручного та самодостатнього інструменту, який забезпечує швидке налаштування сценаріїв, живий моніторинг результатів і автоматичну оцінку відповідності системи визначеним SLA-критеріям.

Мета роботи -алгоритмізація та розробка інструменту для автоматизації тестування продуктивності веб-додатків у розподілених середовищах.

Завдання роботи:

- проаналізувати сучасні підходи та інструменти для навантажувального тестування;
- визначити вимоги до інструменту для підтримки автоматизації та розподілених запусків;
- розробити архітектуру системи PerfBench (Control Plane, інтерфейс користувача, агенти, демо-ціль);

- реалізувати основні модулі: рушії навантаження, збір метрик, SLA-аналіз, звітність;
- створити інтерфейс користувача для зручного опису сценаріїв та перегляду результатів;
- перевірити роботу інструменту на експериментальних сценаріях.

Об’єкт дослідження -процес навантажувального тестування веб-додатків у розподілених середовищах.

Предмет дослідження -методи алгоритмізації, проектування та реалізації програмних інструментів для автоматизації тестування продуктивності.

Практичне значення роботи полягає у створенні програмного продукту PerfBench, який може бути використаний інженерами продуктивності, тестувальниками та командами розробки для швидкої організації навантажувальних тестів, аналізу результатів та підвищення надійності веб-додатків.

1. ПОСТАНОВКА ЗАДАЧІ

Основною метою даного проєкту є розробка інструменту для автоматизації тестування продуктивності веб-додатків у розподілених середовищах. Такий інструмент має спростити процес створення та запуску навантажувальних сценаріїв, забезпечити збір і візуалізацію метрик у режимі реального часу, а також підтримати можливість розподіленого виконання для моделювання масштабних умов роботи системи.

Основні завдання:

1. Аналіз предметної області:
    - дослідити сучасні методи та підходи до тестування продуктивності;
    - визначити ключові метрики (latency, throughput, error rate, SLA-пороги).
2. Проєктування архітектури системи:
    - розробити структуру інструменту з Control Plane, веб-інтерфейсом, демо-ціллю та опційними агентами;
    - визначити модель даних для сценаріїв, запусків та метрик.
3. Розробка сценаріїв навантаження:
    - забезпечити підтримку steady, ramp та spike профілів;
    - реалізувати flow-сценарії з кроками та екстракцією змінних;
    - додати можливість опису traffic mix для різних маршрутів.
4. Реалізація рушіїв навантаження:
    - створити вбудований генератор запитів (native engine);
    - забезпечити інтеграцію з k6 як додатковим рушієм.
5. Розробка інтерфейсу користувача:
    - створити UI для формування сценаріїв, запуску тестів та перегляду результатів;
    - забезпечити відображення живих метрик та підсумкових звітів.
6. Реалізація розподілених запусків:
    - розробити протокол взаємодії агентів з Control Plane (реєстрація, heartbeat, claim, complete);

- реалізувати механізм шардінгу навантаження між агентами.

7. Аналітика та звітність:
   - забезпечити формування summary з p50/p95/p99, throughput та error rate;
   - реалізувати експорт результатів у CSV та HTML;
   - додати baseline-порівняння запусків.
8. Документація та демонстрація:
   - створити супровідну документацію для користувачів;
   - підготувати демонстраційні сценарії (steady, spike+jitter, flow, traffic mix).

Головними вимогами до інструменту є стабільність роботи, зручність використання, підтримка гнучкого опису сценаріїв та можливість масштабування у розподілених середовищах. PerfBench має забезпечити інженерам продуктивності швидке налаштування тестів, прозорий збір метрик та ефективний аналіз результатів.

2. ІНФОРМАЦІЙНИЙ ОГЛЯД

2.1. Підходи та методи тестування продуктивності

Тестування продуктивності займає особливе місце серед методів забезпечення якості програмного забезпечення. Якщо функціональні тести дають відповідь на запитання «чи правильно працює система», то навантажувальні перевірки дозволяють з’ясувати «як саме вона працює у реальних умовах, коли користувачів багато і їхня активність непередбачувана».

Основна ідея цього типу тестування полягає у створенні контрольованого навантаження на систему та подальшому аналізі її поведінки. Наприклад, якщо веб-додаток розрахований на роботу з тисячею користувачів одночасно, завданням тестування є перевірка того, чи справді він витримає таку кількість з’єднань, наскільки швидко відповідатиме на запити і як змінюватиметься якість обслуговування при перевищенні заявленої межі.

У практиці використовують кілька підходів. Один із найпоширеніших - навантажувальне тестування, коли система перевіряється в умовах, максимально наближених до звичайної експлуатації. Наприклад, інтернет-магазин може бути протестований за сценарієм, де одночасно сотні користувачів переглядають товари, додають їх у кошик і оформлюють замовлення. Це дозволяє виявити, наскільки швидко сайт обробляє типові бізнес-процеси.

Інший підхід -стрес-тестування, коли навантаження штучно збільшується доти, поки система не почне працювати з відмовами або значною деградацією. Такий метод дає змогу визначити граничні можливості програмного забезпечення та оцінити, наскільки воно здатне відновитися після перевантаження. Як приклад можна навести ситуацію з онлайн-кінотеатром, який у день прем’єри фільму зазнає різкого напливу глядачів -стрес-тест дозволяє спрогнозувати, чи впорається платформа з подібним піковим навантаженням.

Окрему увагу приділяють так званим «спайк-тестам», коли навантаження різко зростає у короткий проміжок часу. Це актуально для веб-додатків, які можуть зазнати миттєвих стрибків трафіку, наприклад, під час розпродажу в інтернет-

магазині або в період публікації резонансної новини. Якщо система не готова до подібних стрибків, користувачі можуть стикнутися з недоступністю сервісу навіть за відносно невеликої загальної кількості відвідувачів.

Ще один поширений метод -тестування на витривалість, коли додаток працює під стабільним навантаженням упродовж тривалого часу. Такий підхід дозволяє виявити приховані проблеми, які не проявляються під час коротких тестів, наприклад, витоки пам’яті або зростання часу відповіді через накопичення внутрішніх помилок. Для банківських або телекомунікаційних систем подібні тести мають особливе значення, адже стабільність роботи протягом тижнів чи місяців є критичною вимогою.

Усі зазначені підходи зазвичай використовуються разом, адже лише їх поєднання дає повне уявлення про стан продуктивності системи. У реальних проектах спочатку формують базові показники, які виступають орієнтиром (baseline), а потім порівнюють їх із результатами після внесення змін у код чи інфраструктуру. Наприклад, якщо нова версія веб-додатку показує значне зростання часу відповіді на тому ж навантаженні, це сигнал до необхідності оптимізації.

Таким чином, методи тестування продуктивності забезпечують розробників і бізнес-аналітиків даними для прийняття рішень щодо масштабування системи, оптимізації коду та підвищення надійності сервісів. Без них складно гарантувати, що веб-додаток залишатиметься стабільним у критичні моменти, коли від цього залежить довіра користувачів та фінансовий результат компанії.

2.2. Метрики та критерії оцінки (SLA, p95, throughput, error rate)

Ефективність будь-якого навантажувального тестування визначається не лише самим процесом генерації трафіку, але й тим, які саме показники збираються та аналізуються. Метрики продуктивності дозволяють оцінити, наскільки система відповідає очікуванням користувачів і чи може вона виконати умови угод про рівень сервісу (SLA). Якщо для функціональних тестів достатньо отримати «так» або «ні» на запитання про правильність роботи, то у випадку з продуктивністю

необхідна кількісна оцінка.

Однією з ключових метрик є час відгуку системи. Він показує, скільки мілісекунд проходить від моменту надсилання запиту до отримання відповіді. Оскільки середнє значення може бути оманливим, у практиці зазвичай використовують перцентилі. Найчастіше застосовується показник p95, який означає, що 95 % запитів виконалися швидше за вказаний час. Наприклад, якщо p95 дорівнює 800 мс, то лише 5 % запитів перевищили цю межу. Це дає більш реалістичне уявлення про роботу системи, адже навіть незначна кількість повільних відповідей може істотно вплинути на користувацький досвід. У деяких випадках застосовують і більш жорсткі показники, як-от p99 або p99.9, коли йдеться про критичні сервіси на кшталт платіжних систем.

Ще одна важлива характеристика -пропускна здатність або throughput. Вона вимірюється як кількість успішно оброблених запитів за секунду (requests per second). Цей показник дозволяє оцінити, наскільки система здатна масштабуватися під зростання навантаження. Наприклад, якщо веб-додаток стабільно обробляє 500 запитів на секунду, але при переході до 700 починає зростати час відгуку та кількість помилок, це означає, що межа продуктивності досягнута і потрібна оптимізація або розподіл навантаження на додаткові ресурси.

Не менш суттєвою є метрика рівня помилок -error rate. Вона визначається як відсоток невдалих запитів відносно їх загальної кількості. Під помилками зазвичай розуміють відповіді зі статус-кодами HTTP 4xx або 5xx, а також збої мережевих з’єднань. Навіть невелике значення, наприклад 1 %, може бути критичним для фінансових сервісів, де кожна невдала транзакція призводить до втрати грошей чи довіри клієнта.

Усі ці метрики зазвичай об’єднуються у систему критеріїв, яка визначає, чи відповідає додаток встановленим SLA. Наприклад, компанія може зафіксувати у договорі з клієнтами, що p95 часу відгуку не перевищуватиме 500 мс, throughput залишатиметься на рівні не менше ніж 300 запитів на секунду, а рівень помилок не перевищить 0,5 %. Якщо результати тестування підтверджують ці умови, система вважається такою, що відповідає вимогам. Якщо ж хоча б один показник виходить

за межі, необхідно проводити оптимізацію -змінювати архітектуру, масштабувати інфраструктуру або перерозподіляти навантаження.

Таким чином, метрики продуктивності є основою для кількісної оцінки якості роботи веб-додатків. Вони дозволяють не тільки виявляти проблеми, але й вимірювати прогрес після оптимізацій. Без чітких числових критеріїв будь-яке тестування перетворюється на суб’єктивну оцінку, тоді як SLA і формалізовані показники створюють об’єктивний стандарт, за яким можна судити про готовність системи до реальної експлуатації.

2.3. Огляд існуючих інструментів

Одним із найпоширеніших рішень для навантажувального тестування є **Apache JMeter**. Це відкрите програмне забезпечення з багаторічною історією, яке широко використовується у великих компаніях. Jmeter (див. рис. 2.1) підтримує різні протоколи -HTTP, FTP, JDBC, SOAP, JMS та інші, що робить його універсальним інструментом. Основною перевагою є велика кількість готових плагінів і можливість побудови складних сценаріїв через графічний інтерфейс. Проте використання JMeter часто потребує потужних ресурсів на одній машині, а масштабування через розподілені агенти може бути складним у налаштуванні. Крім того, інтерфейс іноді вважають застарілим, що ускладнює роботу початківців.

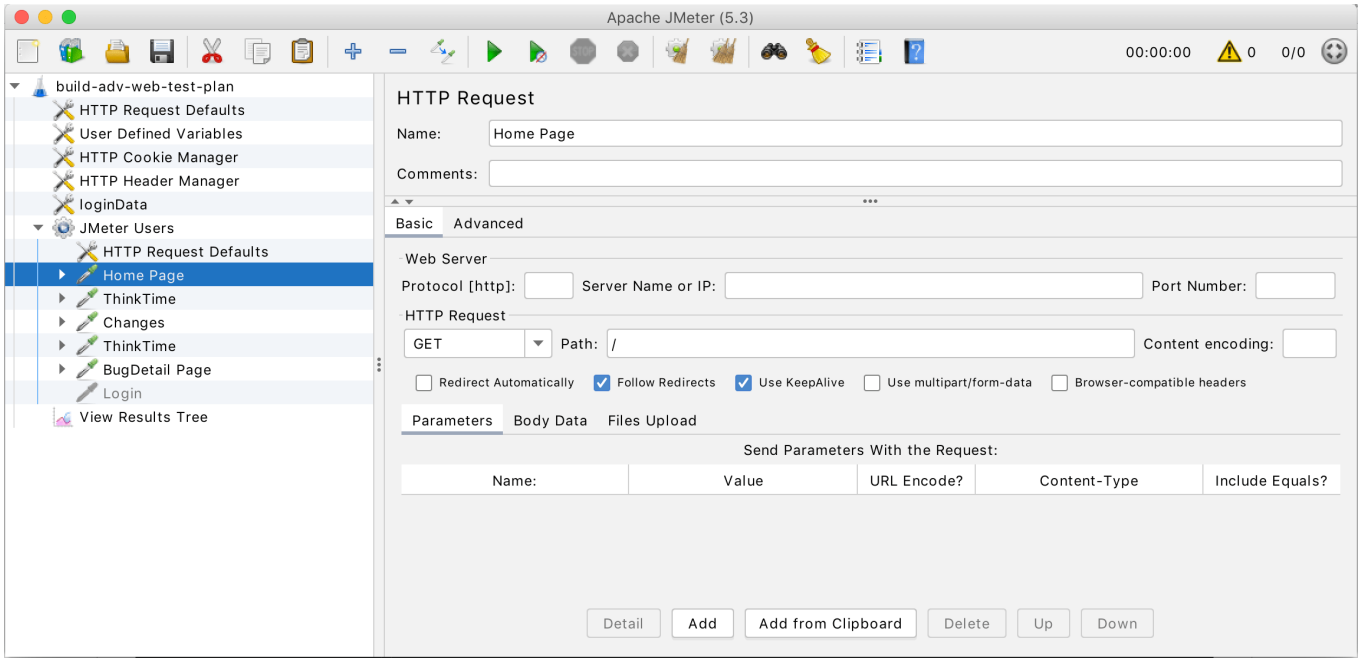


Рисунок 2.1 -Інтерфейс JMeter

Сучасним конкурентом є **k6**, який позиціонується як розробницько-орієнтований інструмент (див. рис. 2.2). Сценарії в k6 описуються мовою JavaScript, що дозволяє легко інтегрувати їх у процес CI/CD. Інженери можуть створювати гнучкі тести з умовами, циклами та змінними, використовуючи знайомий синтаксис. Перевагою є легкість у використанні та підтримка масштабування через хмарний сервіс k6 Cloud. Наприклад, команда може локально запускати базові сценарії, а для великого навантаження -передати їх у хмару. Недоліком є обмежена кількість вбудованих метрик у базовій версії, а також необхідність встановлення самого інструменту на систему.

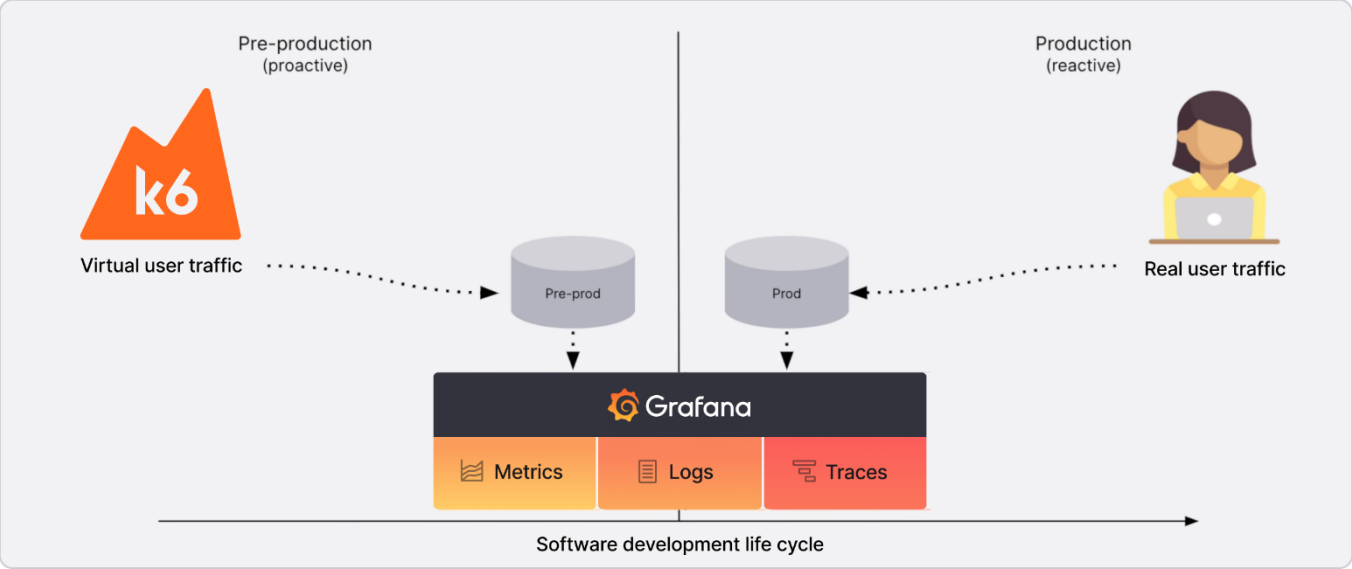
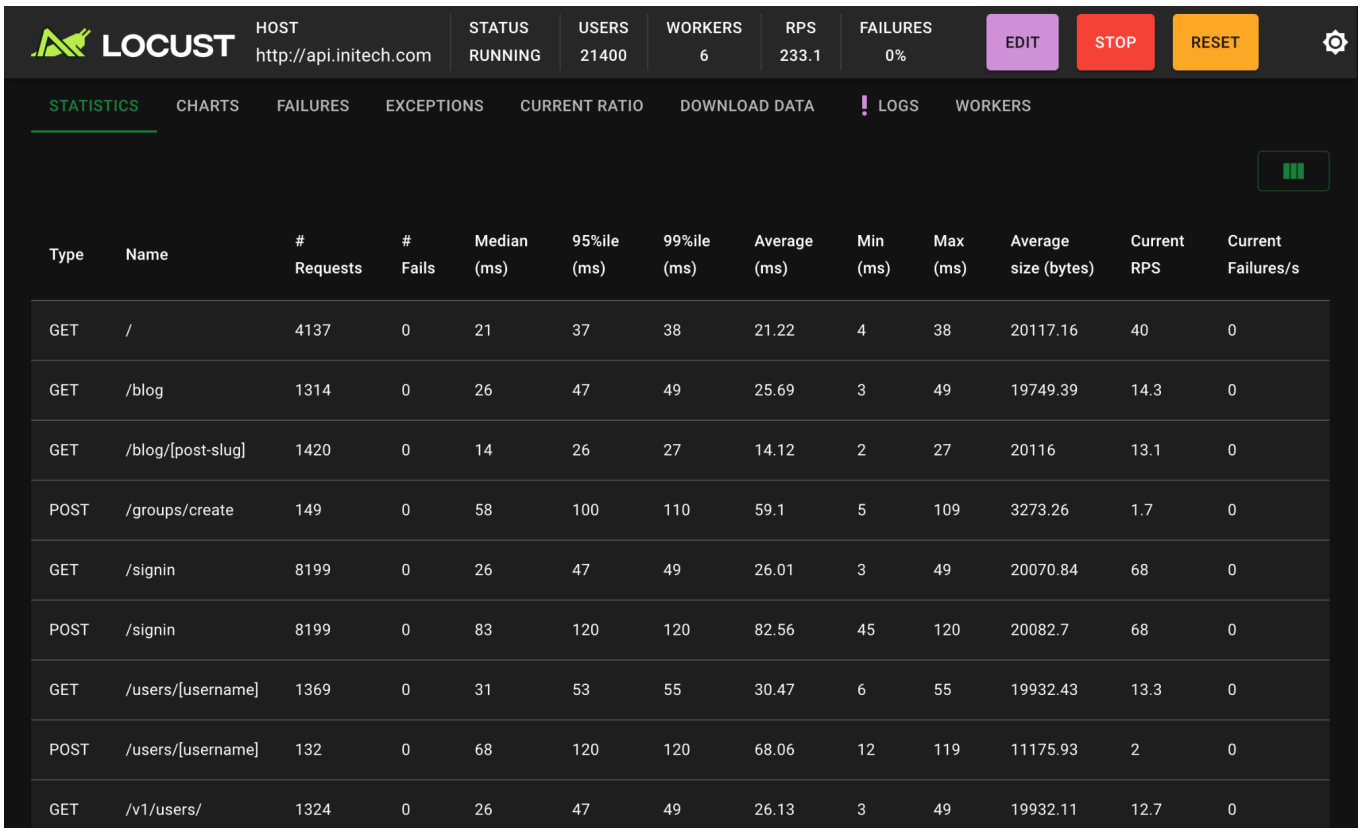


Рисунок 2.2 - Grafana K6

Ще одним популярним рішенням є **Locust** -інструмент на Python (див. рис. 2.3), який дозволяє описувати сценарії у вигляді коду. Його особливістю є простота створення сценаріїв та можливість масштабування через розподіл навантаження на кілька робочих вузлів. Locust має веб-інтерфейс для відстеження результатів у режимі реального часу, що робить його зручним для експериментів. Наприклад, можна швидко перевірити поведінку сайту під час одночасних переходів на головну сторінку та сторінку оформлення замовлення. Водночас, у порівнянні з k6 або JMeter, цей інструмент має менше готових модулів і часто потребує більше ручного налаштування.



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	4137	0	21	37	38	21.22	4	38	20117.16	40	0
GET	/blog	1314	0	26	47	49	25.69	3	49	19749.39	14.3	0
GET	/blog/[post-slug]	1420	0	14	26	27	14.12	2	27	20116	13.1	0
POST	/groups/create	149	0	58	100	110	59.1	5	109	3273.26	1.7	0
GET	/signin	8199	0	26	47	49	26.01	3	49	20070.84	68	0
POST	/signin	8199	0	83	120	120	82.56	45	120	20082.7	68	0
GET	/users/[username]	1369	0	31	53	55	30.47	6	55	19932.43	13.3	0
POST	/users/[username]	132	0	68	120	120	68.06	12	119	11175.93	2	0
GET	/v1/users/	1324	0	26	47	49	26.13	3	49	19932.11	12.7	0

Рисунок 2.3 -Приклад Locust

Варто також згадати **Gatling**, який реалізований на Scala (див. рис. 2.4) і орієнтований на високопродуктивне навантажувальне тестування. Його сильна сторона -висока швидкість генерації запитів навіть на одній машині завдяки асинхронній архітектурі. Gatling підтримує інтеграцію з CI/CD системами та надає зручні HTML-звіти. Наприклад, його часто використовують у телекомунікаційних компаніях, де необхідно моделювати десятки тисяч одночасних з’єднань. Недоліком для деяких команд може стати необхідність знань Scala при створенні складних сценаріїв.

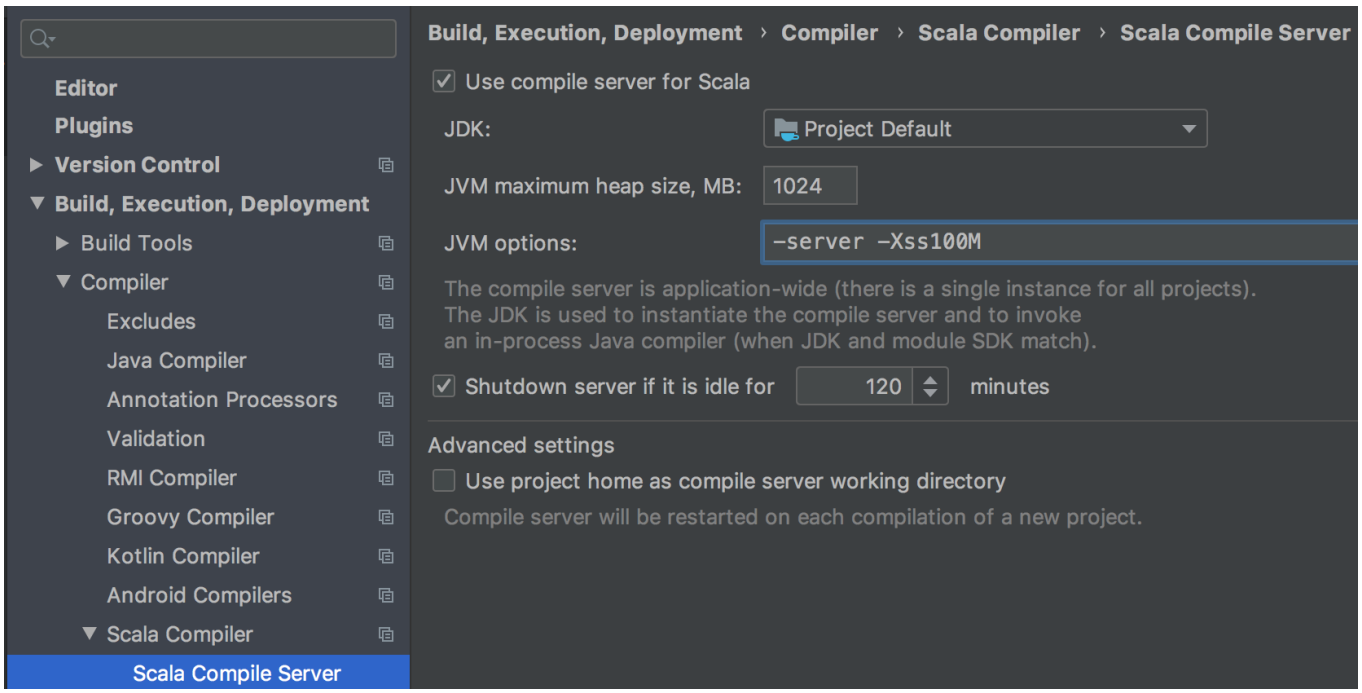


Рисунок 2.4 -Інтерфейс Gatling

Окрему групу складають **комерційні рішення**, як-от LoadRunner або BlazeMeter. Вони орієнтовані на корпоративний сегмент і надають розширені можливості -підтримку великої кількості протоколів, інтеграцію з іншими системами моніторингу, масштабування у хмарі. Наприклад, BlazeMeter дозволяє запускати JMeter-сценарії у хмарній інфраструктурі з тисячами віртуальних користувачів. Проте основним недоліком таких рішень є висока вартість ліцензій, що робить їх малопривабливими для невеликих команд.

Таким чином, існуючі інструменти охоплюють широкий спектр можливостей: від гнучких open-source рішень на кшталт k6 та Locust до корпоративних систем із повною підтримкою та інтеграціями. Вибір залежить від цілей тестування, бюджету та рівня підготовки команди. Разом з тим, кожне з цих рішень має свої обмеження, які створюють нішу для появи нових інструментів, що поєднують простоту, автоматизацію та наочну аналітику. Саме в цій ніші позиціонується розроблений у межах даної роботи інструмент PerfBench.

2.4. Порівняльний аналіз і вибір підходу

Огляд існуючих інструментів показує, що кожне рішення має свої переваги і недоліки, які впливають на вибір команди. Apache JMeter є класичним варіантом для комплексного тестування, але вимагає значних ресурсів і складного налаштування. k6 пропонує більш сучасний підхід із використанням JavaScript для сценаріїв та можливістю інтеграції в CI/CD, але має обмеження у візуалізації та потребує зовнішніх сервісів для масштабування. Locust відрізняється простотою створення сценаріїв на Python, проте його можливості зменшуються у випадку складних сценаріїв. Gatling забезпечує високу продуктивність завдяки асинхронній архітектурі, однак потребує знань Scala. Комерційні рішення надають широкий функціонал і підтримку, але стають недосяжними для невеликих команд через високу вартість.

Щоб краще відобразити ключові характеристики, наведемо порівняльну таблицю найбільш популярних інструментів (див. табл. 2.1)

Інструмент	Переваги	Недоліки	Приклади застосування
Jmeter	Підтримка багатьох протоколів, велика спільнота, графічний інтерфейс	Високе споживання ресурсів, складне масштабування	Тестування веб-додатків у великих корпораціях
k6	Сценарії на JavaScript, інтеграція з CI/CD, хмарне масштабування	Обмежені вбудовані метрики, потребує встановлення	Автоматичні тести продуктивності у DevOps-процесах
Locust	Простота сценаріїв на Python, веб-інтерфейс у реальному часі	Менше готових модулів, обмежена гнучкість	Навантажувальне тестування e-commerce систем
Gatling	Висока продуктивність, HTML-звіти, асинхронність	Необхідність знань Scala, складніший синтаксис	Телекомунікаційні та фінансові системи

Як видно з таблиці, жоден з інструментів не поєднує у собі всі бажані характеристики: простоту налаштування, вбудовану візуалізацію результатів, підтримку розподіленого виконання і при цьому відсутність залежності від дорогих ліцензій. Саме ці обмеження створюють передумови для розробки нових рішень, орієнтованих на швидку автоматизацію та зручний користувацький досвід.

У межах даної дипломної роботи було обрано підхід, який поєднує найкращі риси існуючих інструментів: декларативний опис сценаріїв, автоматичне збирання метрик, відображення результатів у реальному часі, можливість baseline-порівняння та підтримка розподілених агентів. Це стало основою для створення системи PerfBench, яка позиціонується як самодостатнє рішення для тестування продуктивності у розподілених середовищах.

3. ТЕОРЕТИЧНА ЧАСТИНА

3.1. Архітектура інструменту PerfBench

Архітектура системи PerfBench була спроєктована таким чином, щоб забезпечити баланс між простотою використання, гнучкістю сценаріїв і можливістю масштабування у розподілених середовищах. Інструмент складається з кількох основних компонентів, які взаємодіють між собою за допомогою HTTP-інтерфейсів і потоків подій у режимі реального часу.

Центральним елементом системи є Control Plane -серверна частина, реалізована на Node.js з використанням Express і SQLite як бази даних. Саме цей модуль відповідає за управління сценаріями, запуск і координацію тестів, збір метрик і формування звітів. База даних зберігає інформацію про сценарії, запуски, метрики, агентів і результати тестів. Використання SQLite дозволило спростити розгортання, адже система працює без додаткових залежностей і готова до використання «з коробки».

Другим важливим компонентом є веб-інтерфейс, створений за допомогою React і Vite. Він надає користувачеві зручний спосіб створення та редагування сценаріїв, запуску тестів, моніторингу результатів у реальному часі та аналізу завершених запусків. Інтерфейс містить графіки затримок, пропускної здатності, рівня помилок, а також підсумкову таблицю SLA-показників. Особлива увага приділена наочності та простоті взаємодії: навіть користувач, який не має глибокого досвіду роботи з інструментами навантажувального тестування, може швидко налаштувати сценарій і отримати зрозумілий звіт.

Для демонстраційних і навчальних цілей передбачено окремий модуль -Demo Target. Це невеликий веб-сервіс, який імітує роботу реальної системи з параметрами затримки, випадкових помилок і варіативності у часі відповіді. Наприклад, користувач може налаштувати сервіс так, щоб кожен запит оброблявся із затримкою 100 мс, при цьому 5 % запитів завершувалися помилкою. Такий підхід дозволяє проводити експерименти та відпрацьовувати сценарії без підключення до сторонніх систем.

Особливу роль у архітектурі відіграють розподілені агенти. Це легкі процеси на Node.js, які можуть запускатися на різних вузлах мережі та отримувати завдання від Control Plane. Агенти реєструються у системі, надсилають heartbeat-сигнали про свій стан, отримують сценарії для виконання та передають результати тестів назад у центральний сервер. Таким чином, навіть якщо один вузол не здатен згенерувати потрібне навантаження, кілька агентів можуть розподілити його між собою. Наприклад, для моделювання десяти тисяч одночасних користувачів можна підключити кілька агентів на різних серверах, і кожен з них виконуватиме частину навантаження.

Усі компоненти взаємодіють за допомогою REST-запитів і потоків SSE (Server-Sent Events), що дозволяє у реальному часі отримувати метрики і відображати їх у веб-інтерфейсі. Цей підхід спрощує інтеграцію та забезпечує прозорий обмін даними між клієнтом і сервером.

Загалом архітектура PerfBench орієнтована на modular-design: користувач може використовувати лише базову функціональність (локальний запуск із одним рушієм), або ж розширювати систему за рахунок агентів, додаткових рушіїв (наприклад, k6) і інтеграцій. Це робить інструмент придатним як для невеликих експериментів, так і для масштабних перевірок у розподілених середовищах.

3.2. Модель даних та структура сценаріїв

Для того щоб інструмент PerfBench міг забезпечувати повний цикл навантажувального тестування, необхідно було створити модель даних, яка б описувала всі основні сутності системи. Вибір на користь бази даних SQLite пояснюється її простотою, портативністю та відсутністю потреби у зовнішніх залежностях. Це робить інструмент самодостатнім та зручним у розгортанні, адже вся інформація зберігається в одному файлі.

У центрі моделі знаходиться сутність **сценарій**. Сценарій описує, яке саме навантаження буде створюватися, якими параметрами воно характеризується та як будуть розподілятися запити. Він містить назву, опис і головне поле -специфікацію

у форматі JSON. Саме ця специфікація визначає параметри тесту: адресу цілі, метод запиту, кількість одночасних користувачів, тривалість тесту, затримки між запитами, а також SLA-пороги. У більш складних випадках сценарій може включати flow-послідовності, коли спочатку виконується вхід у систему, а далі - отримання даних з використанням токена авторизації.

Другим важливим елементом є **запуск (run)**. Він зберігає інформацію про конкретне виконання сценарію: час початку, завершення, статус, а також примітки та підсумкові результати. Кожен запуск пов’язаний з відповідним сценарієм, що дозволяє відслідковувати історію змін і порівнювати результати різних експериментів.

Для аналізу продуктивності ключову роль відіграють **метрики**. Вони записуються у базу даних у вигляді окремих точок, кожна з яких містить час виконання запиту, затримку у мілісекундах і код відповіді. Такі записи дозволяють будувати графіки продуктивності в реальному часі й формувати агреговані показники, як-от p50, p95 чи середнє значення. Окремо зберігаються **невдалі запити** -вони містять інформацію про статус-код і частину тіла відповіді, що дозволяє розробникам швидко аналізувати причини помилок.

У випадку розподіленого виконання додається ще одна сутність -**агенти**. Це вузли, які виконують частину навантаження. Для кожного агента зберігаються його ідентифікатор, назва, потужність (capacity), статус та час останнього heartbeat. Таким чином Control Plane має змогу відслідковувати актуальний стан інфраструктури. Кожному агенту призначаються окремі завдання (**assignments**), у яких фіксується, яку саме частину тесту він виконує.

Особливе значення для PerfBench має структура сценарію у форматі JSON. Наприклад, простий сценарій може виглядати так:

```
{
  "engine": "native",
  "target": "http://localhost:9090/work?delay=100&fail=0.05",
  "concurrency": 10,
  "requests": 500,
```

```
  "thresholds": { "p95": 800, "errorRatePct": 1 }
}
```

Тут ми визначаємо цільовий ресурс, рівень конкурентності, кількість запитів і допустимі пороги SLA.

Більш складний приклад може включати flow з кількох кроків: авторизація, отримання токена, виконання запиту з параметрами. У такому випадку сценарій описує логіку, яка наближається до реальних дій користувача. Подібні можливості роблять інструмент не просто генератором трафіку, а середовищем для моделювання повних бізнес-процесів.

Таким чином, модель даних PerfBench охоплює всі етапи роботи з навантажувальними тестами -від опису сценарію до збереження детальних метрик і підсумкових результатів. Завдяки цьому користувачі отримують не лише «сухі цифри», а й повну історію запусків, можливість повторного відтворення умов і базу для аналітики.

3.3. Принцип роботи рушіїв навантаження

Рушій навантаження є центральним елементом будь-якого інструменту для тестування продуктивності, адже саме він відповідає за генерацію трафіку та створення навантаження на цільову систему. У PerfBench реалізовано два варіанти рушіїв: власний вбудований механізм і інтеграцію з інструментом k6. Це дозволяє поєднати простоту та контрольованість локального генератора з гнучкістю і потужністю зовнішнього рішення.

Вбудований рушій працює на основі бібліотеки **undici** для Node.js, яка забезпечує асинхронне виконання великої кількості HTTP-запитів. Під час запуску сценарію Control Plane ініціює пул «віртуальних користувачів», які надсилають запити відповідно до заданих параметрів. Наприклад, якщо у сценарії вказано concurrency = 20 та requests = 1000, рушій розподілить навантаження таким чином, що одночасно працюватиме 20 потоків, які поступово виконуватимуть необхідну

кількість запитів. Важливим аспектом є можливість задавати профілі навантаження: steady для стабільного потоку, ramp для поступового зростання та spike для різкого стрибка. Таким чином, навіть базовий рушій дозволяє моделювати типові сценарії використання веб-додатків.

Окрім простих параметрів, вбудований рушій підтримує більш складні сценарії. Flow-моделі дозволяють описувати послідовність кроків, що імітують реальні дії користувача, наприклад: авторизація, отримання токена, завантаження даних. У кожному кроці можуть виконуватися запити з екстракцією змінних із відповіді та подальшою підстановкою їх у наступні запити. Це наближає тестування до реальних бізнес-процесів і робить результати більш практичними для команди розробки.

Інтеграція з **k6** дозволяє розширити можливості PerfBench у тих випадках, коли потрібні масштабні сценарії або специфічні можливості, відсутні у вбудованому рушії. При цьому PerfBench автоматично генерує k6-скрипт на основі описаного сценарію та виконує його за допомогою команди k6 run. Після завершення тесту результати у форматі summary парсяться та додаються у звіт системи. Це дає змогу поєднати зручність роботи через інтерфейс PerfBench із надійністю перевіреного у промислових середовищах інструменту k6. Наприклад, інженер може створити сценарій із mix-трафіком через UI, а фактичне навантаження буде згенероване k6 з точним контролем усіх параметрів.

Ключовою особливістю обох рушіїв є збір метрик у реальному часі. У випадку native-рушія кожен запит записується в базу з часом виконання, статусом і затримкою. У паралельному потоці ці дані агрегуються та надсилаються через SSE у веб-інтерфейс, де користувач бачить графіки середньої затримки, перцентилів, пропускної здатності та рівня помилок. Завдяки цьому можна не чекати завершення тесту, щоб оцінити поведінку системи: достатньо кількох секунд запуску, аби помітити тенденцію.

Таким чином, рушії навантаження у PerfBench забезпечують баланс між простотою та потужністю. Вбудований механізм дозволяє швидко запускати локальні сценарії без додаткових залежностей, тоді як інтеграція з k6 відкриває

шлях до масштабного промислового тестування. Обидва варіанти працюють у рамках єдиної системи, що спрощує роботу користувача та робить інструмент універсальним для різних завдань.

4. ПРАКТИЧНА ЧАСТИНА

4.1. Реалізація бекенду

Бекенд системи PerfBench реалізований на платформі Node.js з використанням TypeScript та фреймворку Express. Основне завдання серверної частини -зберігати сценарії тестування, керувати запуском навантажувальних випробувань, збирати метрики та надавати результати через API. Для збереження даних використовується SQLite, що дозволяє обійтися без додаткової інфраструктури та робить систему придатною для швидкого розгортання.

Архітектурно бекенд складається з кількох модулів:

- index.ts -точка входу, що ініціалізує сервер і підключає маршрути;
- store.ts -модуль роботи з базою даних;
- routes/ -директорія з окремими файлами для сценаріїв, запусків, агентів і пресетів.

Запуск сервера здійснюється у файлі index.ts. Тут створюється екземпляр Express, підключаються middleware для обробки JSON-запитів і CORS, а також визначаються основні маршрути.

```
import express from "express";
import cors from "cors";
import bodyParser from "body-parser";

import scenariosRouter from "./routes/scenarios";
import runsRouter from "./routes/runs";
import agentsRouter from "./routes/agents";
import presetsRouter from "./routes/presets";

const app = express();
app.use(cors());
app.use(bodyParser.json());

// health-check
app.get("/health", (req, res) => {
  res.json({ status: "ok" });
```

```
});

// основні маршрути
app.use("/api/v1/scenarios", scenariosRouter);
app.use("/api/v1/runs", runsRouter);
app.use("/api/v1/agents", agentsRouter);
app.use("/api/v1/presets", presetsRouter);

const PORT = 8080;
app.listen(PORT, () => {
  console.log(`PerfBench backend listening on port ${PORT}`);
});
```

Зберігання даних організоване через SQLite. При першому запуску створюються таблиці для сценаріїв, запусків, метрик і агентів. У файлі store.ts описана логіка підключення до бази та ініціалізації структури:

```
import Database from "better-sqlite3";

export const db = new Database("perfbench.db");

// створення таблиць
db.exec(`
  CREATE TABLE IF NOT EXISTS scenarios (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    description TEXT,
    spec TEXT,
    createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
    updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP
  );

  CREATE TABLE IF NOT EXISTS runs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    scenarioId INTEGER,
    status TEXT,
    startedAt DATETIME,
    finishedAt DATETIME,
    notes TEXT,
```

```
  createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
  updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP
 );

 CREATE TABLE IF NOT EXISTS metrics (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  runId INTEGER,
  createdAt DATETIME,
  elapsedMs INTEGER,
  status INTEGER
 );
`);
```

Для сценаріїв реалізований окремий маршрут scenarios.ts, який дозволяє створювати, отримувати та видаляти записи. Кожен сценарій зберігає JSON-специфікацію, що визначає параметри тестування.

```
import { Router } from "express";
import { db } from "../store";

const router = Router();

// отримати всі сценарії
router.get("/", (req, res) => {
 const rows = db.prepare("SELECT * FROM scenarios").all();
 res.json(rows);
});

// створити новий сценарій
router.post("/", (req, res) => {
 const { name, description, spec } = req.body;
 const stmt = db.prepare("INSERT INTO scenarios (name, description, spec) VALUES (?, ?, ?)");
 const result = stmt.run(name, description, JSON.stringify(spec));
 res.json({ id: result.lastInsertRowid });
});

// видалити сценарій
router.delete("/:id", (req, res) => {
 const stmt = db.prepare("DELETE FROM scenarios WHERE id = ?");
```

```
  stmt.run(req.params.id);
  res.json({ success: true });
});

export default router;
```

Аналогічним чином побудовані маршрути для запусків (runs.ts) та агентів (agents.ts). У випадку запусків реалізована логіка створення нового тесту, обробка зупинки, збирання метрик і формування підсумкових звітів.

Таким чином, бекенд PerfBench виконує роль «керуючого центру», який координує всі процеси: від створення сценаріїв і запуску тестів до збору метрик і формування звітів. Його архітектура побудована так, щоб залишатися максимально простою і водночас забезпечувати всі необхідні можливості для автоматизації тестування продуктивності.

4.2. Реалізація фронтенду

Фронтенд системи PerfBench реалізований з використанням **React** та **TypeScript**, а для швидкої збірки застосовується інструмент Vite. Основне завдання інтерфейсу полягає у тому, щоб надати користувачеві простий та зрозумілий спосіб створення сценаріїв, запуску тестів і перегляду результатів у режимі реального часу.

Структура клієнтської частини включає кілька основних компонентів:

- App.tsx -головний компонент, що відповідає за маршрутизацію між сторінками;
- Templates.tsx -перелік готових шаблонів сценаріїв;
- Guide.tsx -довідкові матеріали з поясненнями;
- RunDetails.tsx -сторінка з результатами запуску, де відображаються графіки та метрики;
- api.ts -модуль взаємодії з бекендом через REST API.

Найважливішим елементом є сторінка створення сценарію. Тут користувач може вказати цільовий URL, метод запиту, параметри конкурентності, тривалість,

профіль навантаження та SLA-пороги. Для цього використовується звичайна форма з контролями React.

```
import { useState } from "react";

export default function ScenarioForm({ onSubmit }: { onSubmit: (data: any) => void }) {
 const [target, setTarget] = useState("");
 const [concurrency, setConcurrency] = useState(10);
 const [requests, setRequests] = useState(100);
 const [p95, setP95] = useState(800);

 const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault();
  onSubmit({
   engine: "native",
   target,
   concurrency,
   requests,
   thresholds: { p95, errorRatePct: 1 }
  });
 };

 return (
  <form onSubmit={handleSubmit} className="form">
   <label>Target URL</label>
   <input type="text" value={target} onChange={e => setTarget(e.target.value)} />

   <label>Concurrency</label>
   <input type="number" value={concurrency} onChange={e => setConcurrency(+e.target.value)} />

   <label>Requests</label>
   <input type="number" value={requests} onChange={e => setRequests(+e.target.value)} />

   <label>P95 threshold (ms)</label>
   <input type="number" value={p95} onChange={e => setP95(+e.target.value)} />

   <button type="submit">Зберегти сценарій</button>
  </form>
 );
```

```
}
```

Щоб отримати метрики у режимі реального часу, фронтенд підключається до **SSE-потоку** бекенду. Це дає можливість малювати графіки затримок, пропускної здатності та рівня помилок без перезавантаження сторінки.

```
import { useEffect, useState } from "react";

export default function RunStream({ runId }: { runId: number }) {
 const [latencies, setLatencies] = useState<number[]>([]);

 useEffect(() => {
  const eventSource = new EventSource(`/api/v1/runs/${runId}/stream`);
  eventSource.addEventListener("metric", (e: any) => {
   const data = JSON.parse(e.data);
   setLatencies(prev => [...prev, data.elapsedMs]);
  });
  return () => eventSource.close();
 }, [runId]);

 return (
  <div>
   <h3>Live Latency</h3>
   <ul>
    {latencies.slice(-10).map((ms, idx) => (
     <li key={idx}>{ms} ms</li>
    ))}
   </ul>
  </div>
 );
}
```

Для відображення аналітики використовуються вбудовані графіки та таблиці. Наприклад, у компоненті **RunDetails.tsx** користувач бачить підсумкові метрики запуску -p50, p95, throughput і рівень помилок. Це дозволяє швидко оцінити, чи пройшла система перевірку за SLA.

```
type Summary = { p50: number; p95: number; throughput: number; errorRate: number };

export default function RunSummary({ data }: { data: Summary }) {
```

```
 return (
  <div className="card">
   <h3>Підсумки запуску</h3>
   <p>p50: {data.p50} ms</p>
   <p>p95: {data.p95} ms</p>
   <p>Throughput: {data.throughput} req/s</p>
   <p>Error Rate: {data.errorRate}%</p>
  </div>
 );
}
```

Завдяки такій організації користувач отримує інтуїтивний інтерфейс: він створює сценарій через форму, запускає тестування та одразу бачить живі графіки і підсумкові показники. Це дозволяє швидко реагувати на виявлені проблеми й порівнювати результати різних запусків.

4.3. Демонстраційна ціль і сценарії запусків

Щоб перевіряти роботу PerfBench без підключення до сторонніх сервісів, у проєкті є невелика «штучна» ціль -HTTP-сервіс, який імітує поведінку реального веб-додатку. Він дозволяє керувати затримками, варіативністю часу відповіді та часткою помилок. Так ми отримуємо контрольоване середовище для експериментів, де можна відтворювати steady, ramp і spike профілі та одразу бачити вплив параметрів на p95, throughput та error rate.

Нижче наведено спрощений варіант server.js для демо-цілі (Express). Маршрут /work приймає delay, jitter і fail, а домашня сторінка транслює агреговані метрики через SSE.

```
const express = require("express");
const app = express();

app.use(express.json());

let stats = {
 total: 0, ok: 0, errors: 0,
 sumLatency: 0, latencies: []
```

```
};

// SSE: транслюємо коротку зведену аналітику кожні ~1с
app.get("/", (req, res) => {
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.flushHeaders();

  const timer = setInterval(() => {
    const n = stats.latencies.length || 1;
    const avg = Math.round(stats.sumLatency / n);
    const p95 = percentile(stats.latencies, 0.95);
    const payload = {
      total: stats.total,
      ok: stats.ok,
      errors: stats.errors,
      errorRate: +(stats.errors / Math.max(1, stats.total) * 100).toFixed(2),
      avgLatency: avg,
      p95
    };
    res.write(`event: summary\ndata: ${JSON.stringify(payload)}\n\n`);
    // обнуляємо ковзне вікно за 10с
    if (stats.latencies.length > 10000) {
      stats.latencies = stats.latencies.slice(-2000);
      stats.sumLatency = stats.latencies.reduce((a, b) => a + b, 0);
    }
  }, 1000);

  req.on("close", () => clearInterval(timer));
});

app.get("/health", (_, res) => res.json({ status: "ok" }));

app.get("/work", async (req, res) => {
  const base = toMs(req.query.delay, 0);      // базова затримка, мс
  const jitter = toMs(req.query.jitter, 0);   // варіація ±jitter, мс
  const fail = Math.min(Math.max(+req.query.fail || 0, 0), 1); // 0..1
```

```
  const extra = jitter ? randInt(-jitter, jitter) : 0;
  const latency = Math.max(0, base + extra);

  const t0 = Date.now();
  await sleep(latency);

  const isFail = Math.random() < fail;
  const elapsed = Date.now() - t0;

  // збір статистики
  stats.total++;
  stats.sumLatency += elapsed;
  stats.latencies.push(elapsed);

  if (isFail) {
    stats.errors++;
    return res.status(500).json({ ok: false, error: "demo fail", elapsed });
  } else {
    stats.ok++;
    return res.json({ ok: true, elapsed, jitterApplied: extra });
  }
});

// утиліти
function sleep(ms) { return new Promise(r => setTimeout(r, ms)); }
function randInt(a, b) { return Math.floor(Math.random() * (b - a + 1)) + a; }
function toMs(v, def) { return Number.isFinite(+v) ? +v : def; }
function percentile(arr, p) {
  if (!arr.length) return 0;
  const sorted = [...arr].sort((a, b) => a - b);
  const idx = Math.min(sorted.length - 1, Math.floor(p * sorted.length));
  return sorted[idx];
}

const PORT = 9090;
app.listen(PORT, () => console.log(`Demo target on :${PORT}`));
```

Такий сервіс зручний для «керованих» експериментів. Наприклад, якщо запустити /work?delay=100&jitter=80&fail=0.05, ми отримаємо середню затримку

близько 100 мс, розкид у межах ±80 мс і ~5% помилок -цього достатньо, аби промоделювати реальний продакшен-трафік з шумом і збоями.

Щоби швидко перевірити демо-ціль з консолі, достатньо кількох запитів:

```
# Базова перевірка доступності
curl http://localhost:9090/health

# 1) Сталий сценарій: 100 мс без помилок
curl "http://localhost:9090/work?delay=100"

# 2) Випадковість у відповідях: ±80 мс до базових 120 мс
curl "http://localhost:9090/work?delay=120&jitter=80"

# 3) Ін'єкція помилок ~10%
curl "http://localhost:9090/work?delay=80&fail=0.1"
```

Приклади сценаріїв для PerfBench

Нижче -кілька готових spec для запуску з UI або через API. Вони демонструють steady, spike і flow підходи, а також traffic-mix.

Steady / SLA-контроль p95≤500 мс

```
{
  "engine": "native",
  "target": "http://localhost:9090/work?delay=100&fail=0.05",
  "method": "GET",
  "concurrency": 10,
  "requests": 500,
  "thinkTimeMs": 0,
  "thresholds": { "p95": 500, "errorRatePct": 1 },
  "profile": "steady"
}
```

Spike + Jitter (демонстрація стрибка навантаження)

```
{
  "engine": "native",
  "target": "http://localhost:9090/work?delay=90&jitter=80&fail=0.02",
  "concurrency": 5,
  "durationMs": 30000,
  "profile": "spike",
  "baseConcurrency": 5,
```

```
  "spikeConcurrency": 40,
  "thresholds": { "p95": 900, "errorRatePct": 2 }
}
```

Traffic Mix (browse / detail / buy) за 30 секунд

```
{
  "engine": "native",
  "concurrency": 12,
  "durationMs": 30000,
  "mix": [
    { "weight": 70, "request": { "target": "http://localhost:9090/work?delay=80", "method": "GET" } },
    { "weight": 20, "request": { "target": "http://localhost:9090/work?delay=120&jitter=50", "method": "GET" } },
    { "weight": 10, "request": { "target": "http://localhost:9090/work?delay=150&fail=0.03", "method": "POST",
      "headers": { "Content-Type": "application/json" }, "body": "{\"id\":\"123\"}" } }
  ],
  "thresholds": { "p95": 800, "errorRatePct": 1.5 }
}
```

Flow: login → data з екстракцією токена

```
{
  "engine": "native",
  "concurrency": 3,
  "durationMs": 15000,
  "steps": [
    {
      "name": "login",
      "request": {
        "target": "http://localhost:9090/work?login=1&delay=120",
        "method": "POST",
        "headers": { "Content-Type": "application/json" },
        "body": "{\"user\":\"u\",\"pass\":\"p\"}"
      },
      "extract": [{ "var": "token", "path": "token" }],
      "thinkTimeMs": 100
    },
    {
      "name": "data",
      "request": {
```

```
      "target": "http://localhost:9090/work?data=1&auth={{token}}&delay=80&jitter=40",
      "method": "GET"
    },
    "thinkTimeMs": 50
  }
 ],
 "iterations": 3,
 "thresholds": { "p95": 850, "errorRatePct": 1 }
}
```

Типові сценарії захисту

На захисті зручно показати прогресію: спочатку простий steady із зеленою SLA-міткою, далі spike із помітним підйомом p95 під час піку, потім flow-сценарій з екстракцією змінної, і нарешті traffic-mix для «близького до реальності» профілю. Усі ці сценарії можна порівняти через baseline-порівняння, щоб продемонструвати різницю між ранами при зміні delay/jitter/fail або параметрів конкурентності.

4.4. Розподілені агенти на практиці

Розподілені агенти в PerfBench -це легкі Node.js-воркери, які під’єднуються до Control Plane, регулярно надсилають heartbeat, отримують призначення (claim), виконують навантаження локально і повертають bulk-метрики. Така схема дозволяє горизонтально масштабувати генерацію трафіку: кілька агентів на різних вузлах паралельно «відпрацьовують» частки одного run.

Нижче -мінімальний, але робочий приклад агента. Він демонструє повний цикл: register → heartbeat → claim → execute → metrics/bulk → complete, підтримує graceful-shutdown і простий backoff на помилках.

```
import fetch from "node-fetch";
import { setTimeout as sleep } from "timers/promises";

// === Налаштування ===
const CONTROL = process.env.CONTROL || "http://localhost:8080/api/v1";
const AGENT_NAME = process.env.AGENT_NAME || `agent-${Math.random().toString(16).slice(2, 7)}`;
const CAPACITY = Number(process.env.CAPACITY || 50); // умовна максимальна concurrency
const HEARTBEAT_MS = 5_000;
```

```
const CLAIM_INTERVAL_MS = 2_000;
const BULK_FLUSH = 100; // надсилати метрики кожні N запитів
let running = true;

// === Стан агента ===
let agentId: number | null = null;

// Graceful shutdown
process.on("SIGINT", () => (running = false));
process.on("SIGTERM", () => (running = false));

// HTTP утиліта з простим retry/backoff
async function http<T = any>(url: string, init?: any, tries = 3, delay = 500): Promise<T> {
 for (let i = 0; i < tries; i++) {
  try {
   const res = await fetch(url, init);
   if (!res.ok) throw new Error(`${res.status} ${res.statusText}`);
   return (await res.json()) as T;
  } catch (e) {
   if (i === tries - 1) throw e;
   await sleep(delay * (i + 1));
  }
 }
 // @ts-ignore
 return null;
}

async function register() {
 const payload = { name: AGENT_NAME, capacity: CAPACITY };
 const data = await http<{ id: number }>(`${CONTROL}/agents/register`, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(payload),
 });
 agentId = data.id;
 console.log(`[agent] registered id=${agentId} name=${AGENT_NAME} cap=${CAPACITY}`);
}
```

```
async function heartbeatLoop() {
  while (running && agentId) {
    try {
      await http(`${CONTROL}/agents/${agentId}/heartbeat`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ id: agentId }),
      });
    } catch (e) {
      console.warn("[agent] heartbeat failed:", (e as Error).message);
    }
    await sleep(HEARTBEAT_MS);
  }
}

type Assignment = {
  id: number;
  runId: number;
  concurrency: number;
  requests?: number;
  durationMs?: number;
  spec: any; // JSON сценарію, спрощено
};

async function claim(): Promise<Assignment | null> {
  if (!agentId) return null;
  try {
    const data = await http<{ assignment?: Assignment }>(`${CONTROL}/agents/${agentId}/claim`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ id: agentId }),
    });
    return data.assignment ?? null;
  } catch (e) {
    console.warn("[agent] claim error:", (e as Error).message);
    return null;
  }
}
```

```
type Metric = { createdAt: number; elapsedMs: number; status: number };
type Failure = { createdAt: number; elapsedMs: number; status: number; body?: string };

// Примітивний HTTP виконавець (без зайвих залежностей)
async function doHttp(req: any): Promise<{ elapsedMs: number; status: number; body?: string }> {
  const { target, method = "GET", headers, body } = req || {};
  const t0 = Date.now();
  try {
    const res = await fetch(target, { method, headers, body });
    const text = await res.text().catch(() => "");
    return { elapsedMs: Date.now() - t0, status: res.status, body: text };
  } catch (e) {
    // мережеві помилки коду немають → умовно 0
    return { elapsedMs: Date.now() - t0, status: 0, body: String(e) };
  }
}

// Виконання конкретного assignment
async function executeAssignment(a: Assignment) {
  const runId = a.runId;
  const spec = a.spec || {};
  const conc = Math.min(a.concurrency ?? 1, CAPACITY);
  const endAt = a.durationMs ? Date.now() + a.durationMs : null;
  let left = a.requests ?? Infinity;

  const metrics: Metric[] = [];
  const failures: Failure[] = [];

  async function flush() {
    if (!metrics.length && !failures.length) return;
    try {
      await http(`${CONTROL}/runs/${runId}/metrics/bulk`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ metrics: metrics.splice(0), failures: failures.splice(0) }),
      });
    } catch (e) {
```

```
    console.warn("[agent] bulk failed:", (e as Error).message);
    // не дропаємо -залишаємо в буферах, спробуємо наступного разу
  }
}

// проста стратегія: паралельні «воркери» женуть або до ліміту запитів, або до часу
const workers = Array.from({ length: conc }, async () => {
  while (running && (left > 0) && (!endAt || Date.now() < endAt)) {
    const reqSpec = pickRequest(spec);
    const res = await doHttp(reqSpec);
    const point: Metric = { createdAt: Date.now(), elapsedMs: res.elapsedMs, status: res.status };
    metrics.push(point);

    if (res.status === 0 || res.status >= 400) {
      failures.push({ createdAt: point.createdAt, elapsedMs: point.elapsedMs, status: res.status });
    }

    if (Number.isFinite(left)) left--;
    if (metrics.length + failures.length >= BULK_FLUSH) await flush();

    const think = spec.thinkTimeMs ? Number(spec.thinkTimeMs) : 0;
    if (think) await sleep(think);
  }
});

await Promise.all(workers);
await flush();

// повідомляємо Control Plane про завершення assignment
try {
  await http(`${CONTROL}/agents/${agentId}/complete/${a.id}`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ notes: `done by ${AGENT_NAME}` }),
  });
} catch (e) {
  console.warn("[agent] complete failed:", (e as Error).message);
}
```

```
}

// Вибір запиту зі spec: або є mix[], або steps[0], або базовий request
function pickRequest(spec: any) {
  if (Array.isArray(spec?.mix) && spec.mix.length) {
    const sum = spec.mix.reduce((s: number, m: any) => s + (m.weight || 0), 0) || 1;
    let r = Math.random() * sum;
    for (const m of spec.mix) {
      r -= m.weight || 0;
      if (r <= 0) return m.request;
    }
    return spec.mix[0].request;
  }
  if (Array.isArray(spec?.steps) && spec.steps.length) {
    return spec.steps[0].request; // у мінімальній версії ігноруємо екстракцію/темплейтинг
  }
  return {
    target: spec.target,
    method: spec.method || "GET",
    headers: spec.headers,
    body: spec.body,
  };
}

async function main() {
  await register();
  heartbeatLoop(); // без await -фоново

  while (running) {
    const a = await claim();
    if (a) {
      console.log(`[agent] got assignment #${a.id} run=${a.runId} conc=${a.concurrency} req=${a.requests ?? "-"} dur=${a.durationMs ?? "-"}`);
      try {
        await executeAssignment(a);
      } catch (e) {
        console.error("[agent] assignment failed:", (e as Error).message);
      }
```

```
    } else {
      await sleep(CLAIM_INTERVAL_MS);
    }
  }
  console.log("[agent] stopped");
}

main().catch(e => {
  console.error(e);
  process.exit(1);
});
```

Запуск такого воркера достатньо простий:

```
# Варіант 1: локально
CONTROL=http://localhost:8080/api/v1 AGENT_NAME=node-a CAPACITY=80 node dist/worker.js

# Варіант 2: кілька агентів на одній машині
AGENT_NAME=a1 node dist/worker.js &
AGENT_NAME=a2 node dist/worker.js &

# Варіант 3: Docker (ескіз)
# docker run -e CONTROL=http://host.docker.internal:8080/api/v1 -e AGENT_NAME=docker-a -e CAPACITY=120 your/agent:latest
```

З боку Control Plane усе виглядає прозоро. Після запуску агентів можна створити run (через UI або API). Якщо активні агенти є, Control Plane «нашардить» завдання у таблицю run_assignments, і кожен агент забере свій шматок. Під час виконання метрики буферизуються та відправляються на ендпоінт POST /api/v1/runs/:id/metrics/bulk, що суттєво зменшує накладні витрати на мережу і запис у БД.

Для швидкої перевірки API з консолі:

```
# Переглянути зареєстрованих агентів і призначення
curl http://localhost:8080/api/v1/agents
curl http://localhost:8080/api/v1/agents/assignments

# Старт ad-hoc run без збереження сценарію
curl -X POST http://localhost:8080/api/v1/runs \
  -H "Content-Type: application/json" \
```

-d
'{"spec":{"engine":"native","target":"http://localhost:9090/work?delay=100","concurrency":40,"requests":4000}}'

У продакшн-середовищі доцільно додати кілька покращень: контроль черги assignment-ів за capacity, повторні спроби доставки bulk з дисковим буфером на випадок відключення мережі, таймаути/обриви довгих запитів і більш просунутий executor для steps з екстракцією/темплейтингом змінних. Проте навіть наведений мінімум дозволяє повністю відпрацювати розподілений запуск і зібрати коректні агрегати p95/throughput/error rate на Control Plane.

4.5. Інструкція для користувача

1. **Головна сторінка (див. рис. 4.1)**

   1.1. Основний інтерфейс

- Після запуску користувач потрапляє на головну сторінку системи PerfBench.
- У верхньому меню доступні основні розділи: **Головна**, **Шаблони**, **Посібник**.
- На сторінці відображається статус бекенду (індикатор «ok» у правому верхньому куті).
- Нижче подано короткий опис можливостей: створення сценаріїв, запуск тестів, аналітика результатів.
- Додатково представлений блок «Швидкий старт (5 кроків)» із поетапною інструкцією, що допомагає новим користувачам швидко розпочати роботу.

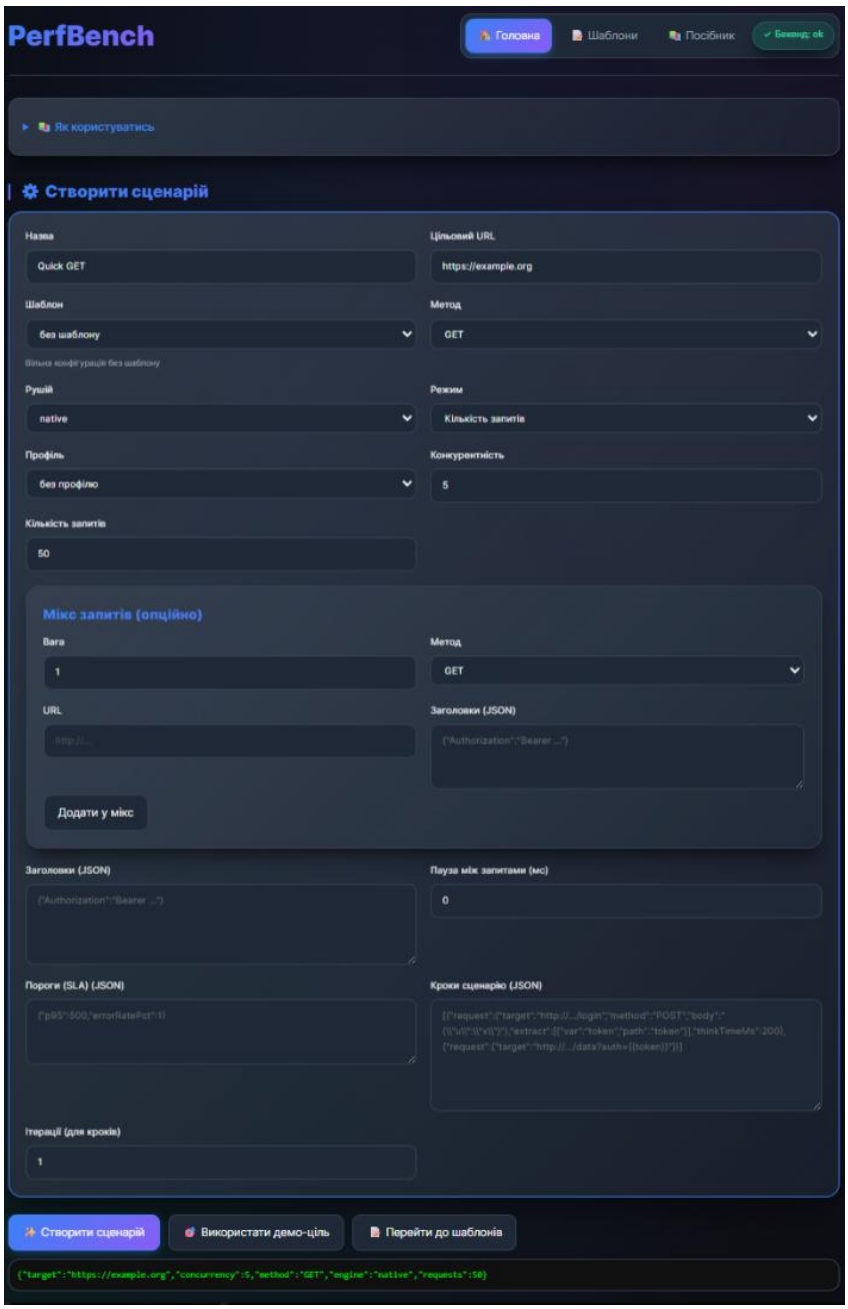


Рисунок 4.1 -Головна сторінка інструменту PerfBench

2. **Створення сценарію (див. рис. 4.2)**

2.1. Конфігурація параметрів

- У розділі «Створити сценарій» користувач задає основні параметри: назву, цільовий URL, метод (GET, POST тощо), рушій (native чи k6), режим (кількість запитів або тривалість).
- Є можливість обрати профіль навантаження: steady, ramp чи spike.
- Додаткові поля дозволяють налаштувати конкурентність, кількість запитів, SLA-пороги (наприклад, p95 ≤ 500 мс), паузу між запитами та flow-кроки.

- Для складних сценаріїв доступний «Мікс запитів» із ваговими коефіцієнтами.

2.2. Дії користувача

- Після заповнення полів можна зберегти сценарій, використати демо-ціль або перейти до шаблонів.
- У нижній частині форми відображається згенерований JSON-специфікація, яку можна редагувати вручну.

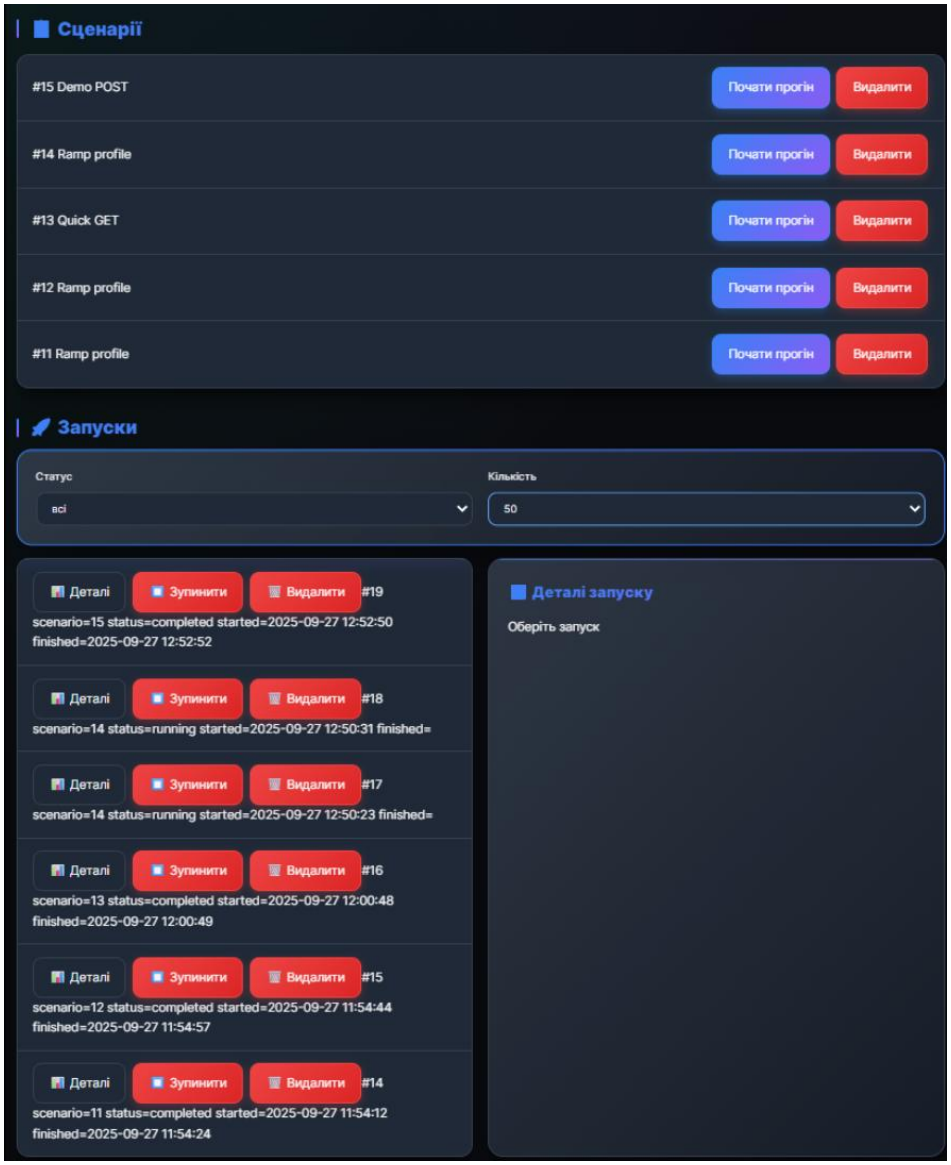


Рисунок 4.2 -Сторінка створення сценарію

3. **Сценарії та запуски (див. рис. 4.3)**

3.1. Робота зі сценаріями

- У розділі «Сценарії» відображається список усіх збережених конфігурацій.

- Біля кожного сценарію доступні кнопки: **Почати прогін** (запуск тесту) та **Видалити**.

3.2. Робота із запусками

- У нижньому блоці показані всі створені запуски із зазначенням їх статусу: running, completed чи failed.
- Кожен запуск має кнопки **Деталі**, **Зупинити** та **Видалити**.
- У панелі праворуч відкриваються деталі конкретного запуску.

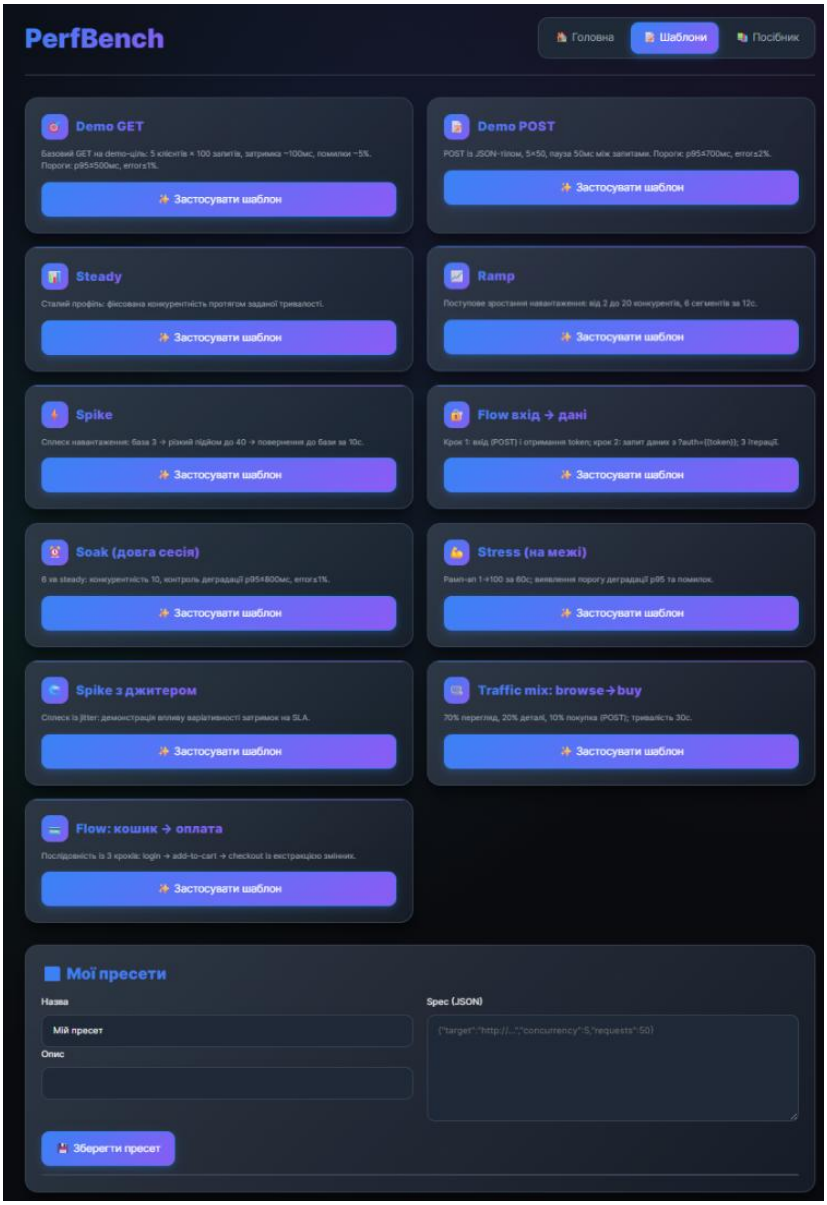


Рисунок 4.3 -Розділ сценаріїв та запусків

4. **Шаблони (див. рис. 4.4)**

4.1. Використання готових пресетів

- Розділ «Шаблони» містить набір типових конфігурацій: Demo GET, Ramp, Spike, Stress, Flow checkout, Traffic mix тощо.
- Кожен шаблон супроводжується описом: наприклад, Spike демонструє різкий стрибок навантаження, а Traffic mix моделює поведінку реальних користувачів.
- Користувач може застосувати шаблон натисканням кнопки «Застосувати шаблон».

4.2. Створення власних пресетів

- У нижній частині сторінки є форма для збереження власних сценаріїв як пресетів, із назвою, описом і JSON-специфікацією.

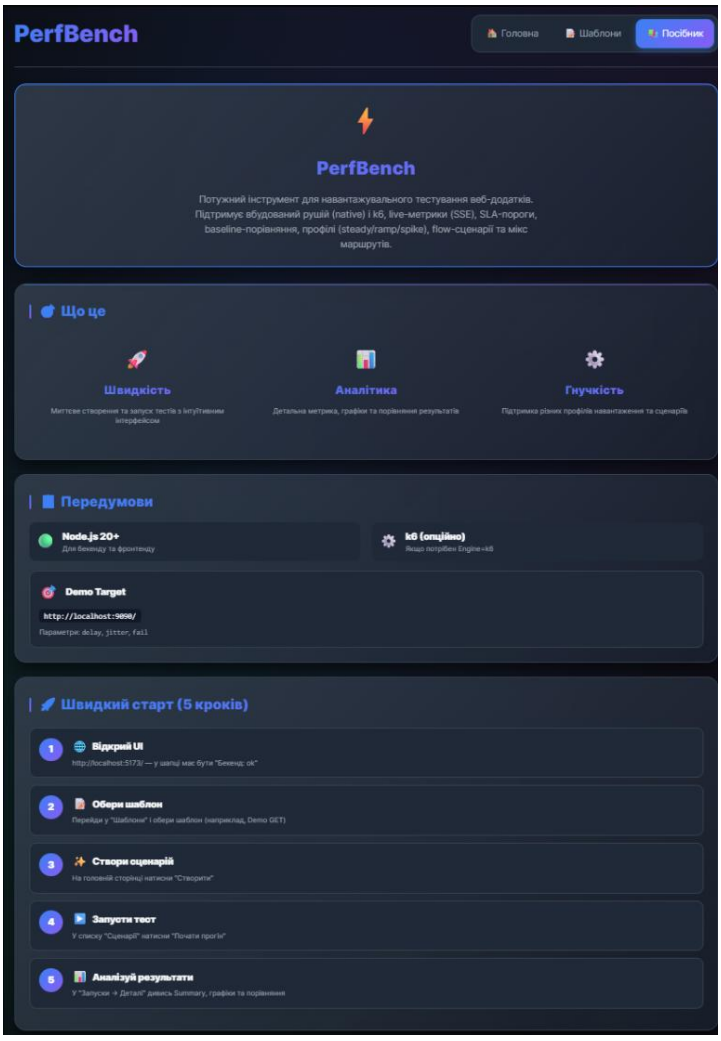


Рисунок 4.4 -Сторінка шаблонів

5. **Посібник користувача (див. рис. 4.5)**

5.1. Огляд функцій

- Розділ «Посібник» містить загальний опис можливостей PerfBench: швидке створення сценаріїв, аналітика результатів, підтримка різних профілів навантаження.
- Тут наведено передумови для запуску (Node.js, demo target, k6) та коротку інструкцію з використання.

5.2. Швидкий старт

- Окремим блоком представлено 5 кроків для запуску тесту: від відкриття UI до аналізу результатів.

Рисунок 4.5 -Посібник користувача

6. **Деталі запуску (див. рис. 4.6)**

6.1. Аналітика результатів

- Сторінка деталей запуску містить ключові метрики: p50, p95, p99, throughput, кількість помилок та error rate.
- SLA-пороги візуально позначаються зеленим (успішно) чи червоним (порушено).

6.2. Візуалізація даних

- Графіки відображають зміну затримки, пропускної здатності та частоти помилок у часі.
- Додатково доступна гістограма затримок і список останніх помилкових запитів.
- Результати можна експортувати у формат CSV.

Рисунок 4.6 -Деталі запуску з аналітикою

7. **Демо-ціль (див. рис. 4.7)**

7.1. Призначення

- Демо-ціль -це вбудований сервіс для перевірки роботи інструменту без підключення до зовнішніх систем.
- Він дозволяє задавати параметри: delay (затримка у мс), jitter (варіативність) і fail (ймовірність помилки).

7.2. Інтерфейс

- На сторінці відображаються живі графіки затримки, статистика запитів (кількість успішних і з помилками), середні значення та перцентилі.
- У нижньому блоці наведений список останніх запитів із часом виконання і кодами відповіді.

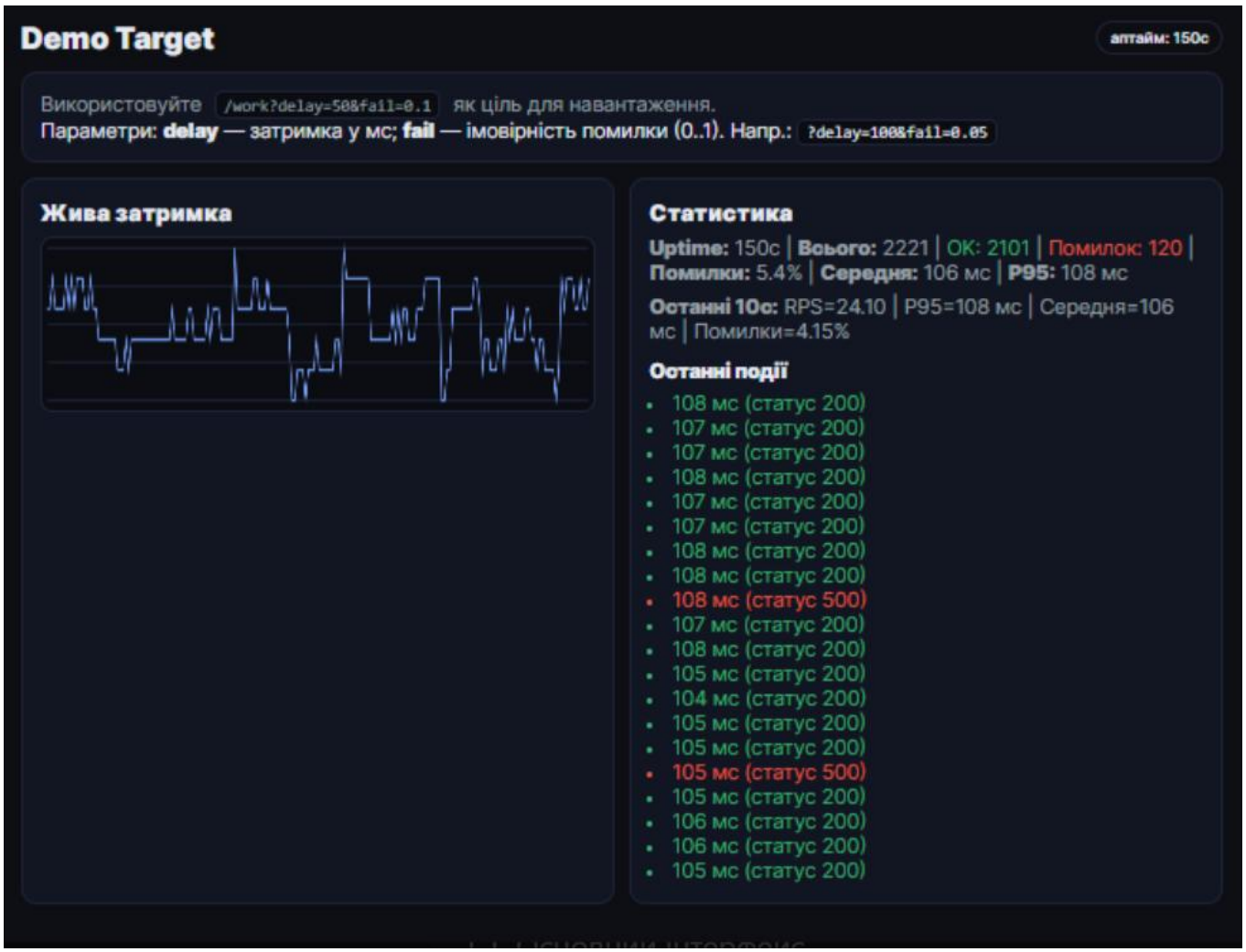


Рисунок 4.7 -Демо-ціль для перевірки навантаження

Таким чином, інтерфейс PerfBench є інтуїтивним і наочним: він дозволяє легко створювати сценарії, запускати навантажувальні тести, аналізувати результати в реальному часі та експортувати дані для подальшої обробки.

ВИСНОВКИ

У ході виконання дипломної роботи було розроблено інструмент PerfBench для автоматизації тестування продуктивності веб-додатків, який поєднує простоту налаштування, гнучкість сценаріїв та підтримку розподілених середовищ. Система дозволяє швидко створювати сценарії навантаження, запускати їх на декількох агентах і аналізувати результати у зручному інтерфейсі.

Основні результати роботи включають:

1. Аналіз існуючих підходів та інструментів
- Проведено огляд сучасних рішень для навантажувального тестування (Apache JMeter, k6, Locust).
- Визначено їхні сильні та слабкі сторони, що дало змогу обґрунтувати створення власного інструменту.
- Сформовано перелік метрик та SLA-критеріїв (p95, throughput, error rate), що використовуються в PerfBench для оцінки продуктивності.
2. Проєктування архітектури
- Створено модульну структуру з поділом на Control Plane (бекенд), UI та агенти.
- Передбачено можливість горизонтального масштабування завдяки підключенню додаткових агентів.
- Забезпечено підтримку як вбудованого рушія (native), так і інтеграції з k6.
3. Реалізація функціональних можливостей
- Розроблено зручний веб-інтерфейс для створення сценаріїв із використанням JSON-специфікацій.
- Реалізовано готові шаблони навантаження: steady, ramp, spike, stress, traffic mix, flow-сценарії.
- Додано модуль збору та візуалізації результатів у реальному часі: затримка, пропускна здатність, частка помилок.
- Реалізовано можливість експорту даних у формат CSV для подальшого аналізу.

4. Розподілене виконання тестів

- Створено легкі агенти на Node.js, які підключаються до Control Plane та виконують призначення.
- Забезпечено відправлення метрик у bulk-режимі, що зменшує накладні витрати.
- Реалізовано heartbeat-механізм і контроль активних агентів.

5. Інтерфейс користувача

- Розроблено інтуїтивний UI з розділами: «Сценарії», «Запуски», «Шаблони», «Посібник».
- Створено інтегрований demo-target, який дозволяє перевірити роботу інструменту без зовнішніх сервісів.
- Запропоновано покрокову інструкцію «Швидкий старт», що робить систему доступною навіть для початківців.

Розроблений інструмент дозволяє:

- моделювати різні профілі навантаження та користувацькі сценарії;
- відстежувати продуктивність веб-додатків у реальному часі;
- швидко виявляти проблеми масштабованості й стабільності;
- автоматизувати процес тестування у розподіленому середовищі.

Розробка підтвердила ефективність використання власного інструменту для навантажувального тестування: PerfBench поєднав простоту інтерфейсу, можливість масштабування та автоматизацію процесу, що робить його корисним як для розробників, так і для інженерів із забезпечення якості.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Apache JMeter. User Manual. Інтернет-доступ: https://jmeter.apache.org/usermanual/

2. Grafana Documentation. Metrics, Dashboards and Visualization. Інтернет-доступ: https://grafana.com/docs/

3. Prometheus Documentation. Monitoring System and Time Series Database. Інтернет-доступ: https://prometheus.io/docs/

4. Node.js Documentation. Інтернет-доступ: https://nodejs.org/en/docs/

5. k6 Load Testing Tool. Documentation. Інтернет-доступ: https://k6.io/docs/

6. Locust Documentation. Scalable User Load Testing. Інтернет-доступ: https://docs.locust.io/

7. Docker Documentation. Build and Run Containers. Інтернет-доступ: https://docs.docker.com/

8. PostgreSQL Documentation. Database System. Інтернет-доступ: https://www.postgresql.org/docs/

9. React Documentation. A JavaScript library for building user interfaces. Інтернет-доступ: https://react.dev/

10. TypeScript Documentation. Strongly typed JavaScript. Інтернет-доступ: https://www.typescriptlang.org/docs/

11. Fetch API Documentation. Інтернет-доступ: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

12. Apache Kafka Documentation. Distributed Streaming Platform. Інтернет-доступ: https://kafka.apache.org/documentation/

13. Ольховська О. В. Методичні рекомендації до виконання кваліфікаційної роботи для студентів спеціальності 122 Комп’ютерні науки освітня програма «Комп’ютерні науки» ступеня бакалавра / О. В. Ольховська, О. О. Черненко. - Полтава : ПУЕТ, 2024. -67 с. -1 електрон. опт. диск (CVD-ROM).

ДОДАТОК А.

```
import express = require('express')
import cors = require('cors')
import { Database } from 'better-sqlite3'
import createDb = require('./store')
import { scenariosRouter } from './routes/scenarios'
import { runsRouter } from './routes/runs'
import { presetsRouter } from './routes/presets'
import { agentsRouter } from './routes/agents'

const app = express()
app.use(cors())
app.use(express.json({ limit: '2mb' }))

let db: Database

app.get('/health', (_req, res) => {
  res.json({ status: 'ok' })
})

app.use('/api/v1/scenarios', scenariosRouter(() => db))
app.use('/api/v1/runs', runsRouter(() => db))
app.use('/api/v1/presets', presetsRouter(() => db))
app.use('/api/v1/agents', agentsRouter(() => db))

const port = process.env.PORT || 8080

async function start() {
  db = createDb('perfbench.db')
  const server = app.listen(port, () => {
    console.log(`server listening on http://localhost:${port}`)
  })
  process.on('SIGINT', () => server.close())
}

start().catch(err => {
  console.error(err)
```

```
  process.exit(1)
})
```

backend/src/store.ts

```typescript
import DatabaseConstructor, { Database } from 'better-sqlite3'

export = function createDb(path: string): Database {
  const db = new DatabaseConstructor(path)
  db.pragma('journal_mode = WAL')
  db.exec(`
    create table if not exists scenarios (
      id integer primary key autoincrement,
      createdAt text not null default (datetime('now')),
      updatedAt text not null default (datetime('now')),
      name text not null,
      description text,
      spec text not null
    );
    create table if not exists runs (
      id integer primary key autoincrement,
      createdAt text not null default (datetime('now')),
      updatedAt text not null default (datetime('now')),
      scenarioId integer not null,
      status text not null,
      startedAt text,
      finishedAt text,
      notes text,
      foreign key (scenarioId) references scenarios(id)
    );
    create table if not exists metrics (
      id integer primary key autoincrement,
      createdAt text not null default (datetime('now')),
      runId integer not null,
      elapsedMs integer not null,
      status integer not null,
```

```
  foreign key (runId) references runs(id)
);
create table if not exists failures (
  id integer primary key autoincrement,
  createdAt text not null default (datetime('now')),
  runId integer not null,
  status integer not null,
  elapsedMs integer not null,
  body text,
  foreign key (runId) references runs(id)
);
create table if not exists presets (
  id integer primary key autoincrement,
  createdAt text not null default (datetime('now')),
  name text not null,
  description text,
  spec text not null
);
create trigger if not exists scenarios_update_ts after update on scenarios begin
  update scenarios set updatedAt = datetime('now') where id = NEW.id; end;
create trigger if not exists runs_update_ts after update on runs begin
  update runs set updatedAt = datetime('now') where id = NEW.id; end;
create table if not exists agents (
  id integer primary key autoincrement,
  name text not null,
  capacity integer not null default 1,
  lastHeartbeat text not null default (datetime('now')),
  status text not null default 'idle'
);
create table if not exists run_assignments (
  id integer primary key autoincrement,
  runId integer not null,
  agentId integer not null,
  concurrency integer not null default 1,
  requests integer,
  durationMs integer,
  status text not null default 'queued',
  startedAt text,
```

```
    finishedAt text,
    foreign key (runId) references runs(id),
    foreign key (agentId) references agents(id)
  );
 `)
 return db
}

import { Router } from 'express'
import { Database } from 'better-sqlite3'
import { z } from 'zod'

export function scenariosRouter(getDb: () => Database) {
 const r = Router()

 r.get('/', (_req, res) => {
  const rows = getDb().prepare('select * from scenarios order by id desc').all()
  res.json(rows)
 })

 const CreateSchema = z.object({
  name: z.string().min(1),
  description: z.string().optional().default(''),
  spec: z.string().min(2),
 })

 r.post('/', (req, res) => {
  const parsed = CreateSchema.safeParse(req.body)
  if (!parsed.success) return res.status(400).json({ error: parsed.error.message })
  const { name, description, spec } = parsed.data
  const stmt = getDb().prepare('insert into scenarios (name, description, spec) values (?, ?, ?)')
  const info = stmt.run(name, description, spec)
  const row = getDb().prepare('select * from scenarios where id = ?').get(info.lastInsertRowid)
  res.status(201).json(row)
 })

 r.get('/:id', (req, res) => {
  const row = getDb().prepare('select * from scenarios where id = ?').get(+req.params.id)
```

```
    if (!row) return res.status(404).json({ error: 'not found' })
    res.json(row)
  })

  r.delete('/:id', (req, res) => {
    getDb().prepare('delete from scenarios where id = ?').run(+req.params.id)
    res.status(204).end()
  })

  return r
}



import { Router, Request, Response } from 'express'
import { Database } from 'better-sqlite3'
import { z } from 'zod'
import { fetch } from 'undici'
import { EventEmitter } from 'events'
import { spawn } from 'child_process'
import * as fs from 'fs'
import * as os from 'os'
import * as path from 'path'

export function runsRouter(getDb: () => Database) {
  const r = Router()

  r.get('/', (req, res) => {
    const db = getDb()
    const where: string[] = []
    const params: any[] = []
    const status = String(req.query.status || '').trim()
    const scenarioId = Number(req.query.scenarioId || 0)
    const offset = Math.max(0, parseInt(String(req.query.offset ?? '0')) || 0)
    const limit = Math.min(200, Math.max(1, parseInt(String(req.query.limit ?? '50')) || 50))
    if (status) { where.push('status = ?'); params.push(status) }
    if (scenarioId) { where.push('scenarioId = ?'); params.push(scenarioId) }
    const sql = `select * from runs ${where.length?('where '+where.join(' and ')):''} order by id desc limit ? offset ?`
    const rows = db.prepare(sql).all(...params, limit, offset)
```

```
    const total = db.prepare(`select count(*) as c from runs ${where.length?('where '+where.join(' and ')):''}`).get(...params) as any
    res.json({ total: total.c as number, items: rows })
  })

  r.get('/:id', (req, res) => {
    const row = getDb().prepare('select * from runs where id = ?').get(+req.params.id)
    if (!row) return res.status(404).json({ error: 'not found' })
    res.json(row)
  })

  r.delete('/:id', (req, res) => {
    const runId = +req.params.id
    const db = getDb()
    db.prepare('delete from metrics where runId = ?').run(runId)
    db.prepare('delete from failures where runId = ?').run(runId)
    db.prepare('delete from runs where id = ?').run(runId)
    res.status(204).end()
  })

  const StartSchema = z.object({ scenarioId: z.number() })

  r.post('/', async (req, res) => {
    const parsed = StartSchema.safeParse(req.body)
    if (!parsed.success) return res.status(400).json({ error: parsed.error.message })
    const { scenarioId } = parsed.data
    const db = getDb()
      const info = db.prepare("insert into runs (scenarioId, status, startedAt) values (?, 'running', datetime('now'))").run(scenarioId)
    const runId = Number(info.lastInsertRowid)

    const agents = db.prepare('select * from agents order by id asc').all() as any[]
    if (agents && agents.length > 0) {
      const scenario = db.prepare('select * from scenarios where id = ?').get(scenarioId) as any
      let spec: any = {}
      try { spec = JSON.parse(scenario.spec) } catch {}
      const num = agents.length
      if (spec && spec.requests) {
```

```
      const per = Math.max(1, Math.floor(spec.requests / num))
      let remaining = spec.requests
      for (const a of agents) {
        const r = Math.min(per, remaining)
        if (r <= 0) break
         db.prepare("insert into run_assignments (runId, agentId, requests, concurrency, status) values (?, ?, ?, ?, 'queued')").run(runId, a.id, r, Math.max(1, Math.floor((spec.concurrency||num)/num)))
        remaining -= r
      }
    } else {
      const perC = Math.max(1, Math.floor((spec.concurrency || num) / num))
      for (const a of agents) {
        db.prepare("insert into run_assignments (runId, agentId, durationMs, concurrency, status) values (?, ?, ?, ?, 'queued')").run(runId, a.id, Math.max(1000, spec.durationMs || 5000), perC)
      }
    }
    db.prepare("update runs set status='running' where id = ?").run(runId)
  } else {
    setImmediate(() => executeRun(db, runId).catch(err => console.error('run error', err)))
  }

  res.status(202).json(db.prepare('select * from runs where id = ?').get(runId))
 })

 r.post('/:id/stop', (req: Request, res: Response) => {
  const runId = +req.params.id
  const ctrl = runControllers.get(runId)
  if (!ctrl) return res.status(404).json({ error: 'run not found or already finished' })
  ctrl.abort()
  res.json({ ok: true })
 })

 r.get('/:id/metrics', (req: Request, res: Response) => {
  const runId = +req.params.id
  const offset = Math.max(0, parseInt(String(req.query.offset ?? '0')) || 0)
  const limit = Math.min(5000, Math.max(1, parseInt(String(req.query.limit ?? '200')) || 200))
  const items = getDb().prepare('select id, createdAt, elapsedMs, status from metrics where runId = ? order by id asc limit ? offset ?').all(runId, limit, offset)
```

```
  const totalRow = getDb().prepare('select count(*) as c from metrics where runId = ?').get(runId) as any
  res.json({ total: totalRow.c as number, items })
 })

 r.get('/:id/metrics.csv', (req: Request, res: Response) => {
  const runId = +req.params.id
  const rows = getDb().prepare('select id, createdAt, elapsedMs, status from metrics where runId = ? order by id asc').all(runId) as any[]
  res.setHeader('Content-Type', 'text/csv; charset=utf-8')
  res.setHeader('Content-Disposition', `attachment; filename="run-${runId}-metrics.csv"`)
  res.write('id,createdAt,elapsedMs,status\n')
  for (const r of rows) {
   res.write(`${r.id},${r.createdAt},${r.elapsedMs},${r.status}\n`)
  }
  res.end()
 })

  const Bulk = z.object({ points: z.array(z.object({ elapsedMs: z.number().int().nonnegative(), status: z.number().int(), createdAt: z.string().optional() })) })
 r.post('/:id/metrics/bulk', (req: Request, res: Response) => {
  const parsed = Bulk.safeParse(req.body)
  if (!parsed.success) return res.status(400).json({ error: parsed.error.message })
  const db = getDb()
  const stmt = db.prepare("insert into metrics (runId, createdAt, elapsedMs, status) values (?, ?, ?, ?)")
  const now = new Date().toISOString()
  const runId = +req.params.id
  const tx = db.transaction((arr: typeof parsed.data.points) => {
   for (const p of arr) {
    stmt.run(runId, p.createdAt || now, p.elapsedMs, p.status)
   }
  })
  tx(parsed.data.points)
  res.json({ ok: true, inserted: parsed.data.points.length })
 })

 r.get('/:id/summary', (req: Request, res: Response) => {
  const runId = +req.params.id
  const db = getDb()
```

```
  const run = db.prepare('select * from runs where id = ?').get(runId) as any
  if (!run) return res.status(404).json({ error: 'not found' })
  const scenario = db.prepare('select * from scenarios where id = ?').get(run.scenarioId) as any
  const rows = db.prepare('select elapsedMs, status from metrics where runId = ? order by id asc').all(runId) as any[]
  const lat = rows.map(r => r.elapsedMs as number).sort((a,b)=>a-b)
  const count = lat.length
  const errors = rows.filter(r => (r.status === 0) || (r.status >= 400)).length
  const p = (q: number) => count ? lat[Math.min(count-1, Math.floor(q * (count-1)))] : 0
  const p50 = p(0.50), p95 = p(0.95), p99 = p(0.99)
  const started = run.startedAt ? new Date(run.startedAt) : new Date()
  const finished = run.finishedAt ? new Date(run.finishedAt) : new Date()
  const durationSec = Math.max(0.001, (finished.getTime() - started.getTime())/1000)
  const throughput = count / durationSec
  const errorRate = count? errors/count:0
  let sla: any = undefined
  try {
   const spec = JSON.parse(scenario?.spec || '{}') as any
   if (spec?.thresholds) {
    sla = {
     p95Target: spec.thresholds.p95 ?? null,
     p95Pass: spec.thresholds.p95 != null ? p95 <= spec.thresholds.p95 : null,
     errorRatePctTarget: spec.thresholds.errorRatePct ?? null,
                errorRatePctPass: spec.thresholds.errorRatePct != null ? (errorRate*100) <= spec.thresholds.errorRatePct : null,
    }
   }
  } catch {}
  res.json({ count, errors, errorRate, p50, p95, p99, throughput, durationSec, sla })
 })

 r.get('/:id/stream', (req: Request, res: Response) => {
  const runId = +req.params.id
  res.setHeader('Content-Type', 'text/event-stream')
  res.setHeader('Cache-Control', 'no-cache')
  res.setHeader('Connection', 'keep-alive')
  res.flushHeaders?.()
  const emitter = getRunEmitter(runId)
```

```
    const onMetric = (data: any) => {
      res.write(`data: ${JSON.stringify(data)}\n\n`)
    }
    emitter.on('metric', onMetric)
    req.on('close', () => {
      emitter.off('metric', onMetric)
      res.end()
    })
  })

  return r
}

type SimpleSpec = {
  target: string
  concurrency?: number
  requests?: number
  method?: string
  body?: string
  headers?: Record<string,string>
  durationMs?: number
  stages?: { durationMs: number; concurrency: number }[]
  engine?: 'native' | 'k6'
  thinkTimeMs?: number
  thresholds?: { p95?: number; errorRatePct?: number }
  steps?: Array<{
    name?: string
    request: { target: string; method?: string; headers?: Record<string,string>; body?: string }
    extract?: Array<{ var: string; path: string }>
    thinkTimeMs?: number
  }>
  iterations?: number
  vars?: Record<string,string>
  profile?: 'steady' | 'ramp' | 'spike'
  startConcurrency?: number
  endConcurrency?: number
  segments?: number
  baseConcurrency?: number
```

```
  spikeConcurrency?: number
  mix?: Array<{ weight: number; request: { target: string; method?: string; headers?: Record<string,string>; body?: string } }>
}

const runControllers = new Map<number, AbortController>()

async function executeRun(db: Database, runId: number) {
  const run = db.prepare('select * from runs where id = ?').get(runId) as any
  const scenario = db.prepare('select * from scenarios where id = ?').get(run.scenarioId) as any
  const spec = JSON.parse(scenario.spec) as SimpleSpec
  const method = (spec.method || 'GET').toUpperCase()

  if (spec.engine === 'k6') {
    await executeK6Run(db, runId, spec, method)
  } else if (spec.steps && spec.steps.length) {
    await executeFlow(db, runId, spec)
  } else if (spec.stages && spec.stages.length) {
    for (const stage of spec.stages) {
      await runForDuration(db, runId, spec.target, method, spec.body, stage.concurrency, stage.durationMs, spec.headers, spec.thinkTimeMs)
    }
  } else if (spec.profile) {
    const stages = deriveStagesFromProfile(spec)
    for (const st of stages) {
      await runForDuration(db, runId, spec.target, method, spec.body, st.concurrency, st.durationMs, spec.headers, spec.thinkTimeMs)
    }
  } else if (spec.durationMs && spec.durationMs > 0) {
    const c = Math.max(1, spec.concurrency ?? 1)
    if (spec.mix && spec.mix.length) {
      await runMixForDuration(db, runId, spec.mix, c, spec.durationMs, spec.thinkTimeMs)
    } else {
      await runForDuration(db, runId, spec.target, method, spec.body, c, spec.durationMs, spec.headers, spec.thinkTimeMs)
    }
  } else {
    const concurrency = Math.max(1, spec.concurrency ?? 1)
```

```
    const requests = Math.max(1, spec.requests ?? 1)
    const queue = Array.from({ length: requests }, (_, i) => i)
    async function workerRequests() {
      while (queue.length) {
        const _ = queue.pop(); if (_ === undefined) break
        if (spec.mix && spec.mix.length) {
          const r = pickMix(spec.mix)
          const m = (r.method || 'GET').toUpperCase()
          await singleRequest(db, runId, r.target, m, r.body, r.headers)
        } else {
          await singleRequest(db, runId, spec.target, method, spec.body, spec.headers)
        }
      }
    }
    const workers = Array.from({ length: concurrency }, () => workerRequests())
    await Promise.all(workers)
  }

  db.prepare("update runs set status='completed', finishedAt=datetime('now') where id = ?").run(runId)
}

async function runForDuration(db: Database, runId: number, target: string, method: string, body: string |
undefined, concurrency: number, durationMs: number, headers?: Record<string,string>, thinkTimeMs?: number)
{
  const deadline = Date.now() + durationMs
  async function worker() {
    while (Date.now() < deadline) {
      await singleRequest(db, runId, target, method, body, headers)
      if (thinkTimeMs && thinkTimeMs > 0) await new Promise(r => setTimeout(r, thinkTimeMs))
    }
  }
  const workers = Array.from({ length: Math.max(1, concurrency) }, () => worker())
  await Promise.all(workers)
}

async function singleRequest(db: Database, runId: number, target: string, method: string, body?: string,
headers?: Record<string,string>, signal?: AbortSignal) {
  const start = Date.now()
```

```
  try {
    const res = await fetch(target, { method, body, headers, signal })
    const elapsed = Date.now() - start
    db.prepare('insert into metrics (runId, elapsedMs, status) values (?, ?, ?)').run(runId, elapsed, res.status)
    getRunEmitter(runId).emit('metric', { elapsedMs: elapsed, status: res.status })
    if (res.status >= 400) {
      const text = await res.text().catch(()=> '')
        db.prepare('insert into failures (runId, status, elapsedMs, body) values (?, ?, ?, ?)').run(runId, res.status,
elapsed, text.slice(0, 4096))
    } else {
      await res.arrayBuffer()
    }
  } catch {
    const elapsed = Date.now() - start
    db.prepare('insert into metrics (runId, elapsedMs, status) values (?, ?, 0)').run(runId, elapsed)
    getRunEmitter(runId).emit('metric', { elapsedMs: elapsed, status: 0 })
  }
}

async function executeFlow(db: Database, runId: number, spec: SimpleSpec) {
  const controller = new AbortController()
  runControllers.set(runId, controller)
  try {
    const concurrency = Math.max(1, spec.concurrency ?? 1)
    const iterations = Math.max(1, spec.iterations ?? 1)
    let remaining = iterations * concurrency
    const workers = Array.from({ length: concurrency }, () => flowWorker())
    await Promise.all(workers)
  } finally {
    runControllers.delete(runId)
  }

  async function flowWorker() {
    const vars: Record<string,string> = Object.assign({}, spec.vars || {})
    while (remaining > 0 && !controller.signal.aborted) {
      remaining--
      for (const step of spec.steps || []) {
        const m = (step.request.method || 'GET').toUpperCase()
```

```
      const url = template(step.request.target, vars)
      const body = step.request.body ? template(step.request.body, vars) : undefined
       const headers = step.request.headers ? Object.fromEntries(Object.entries(step.request.headers).map(([k,v]) => [k, template(v, vars)])) : undefined
      const start = Date.now()
      try {
        const res = await fetch(url, { method: m, body, headers, signal: controller.signal })
        const elapsed = Date.now() - start
        db.prepare('insert into metrics (runId, elapsedMs, status) values (?, ?, ?)').run(runId, elapsed, res.status)
        getRunEmitter(runId).emit('metric', { elapsedMs: elapsed, status: res.status })
        let text: string | undefined
        if (res.status >= 400) {
          text = await res.text().catch(()=> '')
           db.prepare('insert into failures (runId, status, elapsedMs, body) values (?, ?, ?, ?)').run(runId, res.status, elapsed, (text||'').slice(0,4096))
        } else {
          text = await res.text().catch(()=> '')
        }
        if (step.extract && text) {
          const parsed = safeJson(text)
          for (const ex of step.extract) {
            const val = getByPath(parsed, ex.path)
            if (val != null) vars[ex.var] = String(val)
          }
        }
      } catch {
        const elapsed = Date.now() - start
        db.prepare('insert into metrics (runId, elapsedMs, status) values (?, ?, 0)').run(runId, elapsed)
        getRunEmitter(runId).emit('metric', { elapsedMs: elapsed, status: 0 })
      }
      const tt = step.thinkTimeMs ?? spec.thinkTimeMs
      if (tt && tt > 0) await new Promise(r => setTimeout(r, tt))
      if (controller.signal.aborted) break
    }
   }
  }
}
```

```
function template(str: string, vars: Record<string,string>) {
  return str.replace(/\{\{(.*?)\}\}/g, (_, k) => vars[k.trim()] ?? '')
}

function safeJson(text: string) { try { return JSON.parse(text) } catch { return {} } }
function getByPath(obj: any, path: string) {
  if (!obj || !path) return undefined
  const parts = path.split('.').filter(Boolean)
  let cur: any = obj
  for (const p of parts) { if (cur && typeof cur === 'object' && p in cur) cur = cur[p]; else return undefined }
  return cur
}

function deriveStagesFromProfile(spec: SimpleSpec): Array<{ durationMs: number; concurrency: number }> {
  const total = Math.max(1000, spec.durationMs ?? 0)
  const segs = Math.max(1, spec.segments ?? 5)
  if (spec.profile === 'steady') {
    const c = Math.max(1, spec.concurrency ?? 1)
    return [{ durationMs: total, concurrency: c }]
  }
  if (spec.profile === 'ramp') {
    const start = Math.max(1, spec.startConcurrency ?? spec.concurrency ?? 1)
    const end = Math.max(1, spec.endConcurrency ?? start)
    const per = Math.floor(total / segs)
    const stages: Array<{ durationMs:number; concurrency:number }> = []
    for (let i = 0; i < segs; i++) {
      const t = i/(segs-1 || 1)
      const c = Math.round(start + (end - start) * t)
      stages.push({ durationMs: per, concurrency: Math.max(1, c) })
    }
    return stages
  }
  const base = Math.max(1, spec.baseConcurrency ?? spec.concurrency ?? 1)
  const spike = Math.max(base, spec.spikeConcurrency ?? (base * 3))
  const warm = Math.floor(total * 0.4)
  const spikeDur = Math.floor(total * 0.2)
  const cool = total - warm - spikeDur
  return [
```

```
    { durationMs: warm, concurrency: base },
    { durationMs: spikeDur, concurrency: spike },
    { durationMs: cool, concurrency: base },
  ]
}

function pickMix(mix: Array<{ weight: number; request: { target: string; method?: string; headers?: Record<string,string>; body?: string } }>) {
  const total = mix.reduce((a, b) => a + Math.max(0, b.weight || 0), 0) || 1
  let r = Math.random() * total
  for (const item of mix) {
    r -= Math.max(0, item.weight || 0)
    if (r <= 0) return item.request
  }
  return mix[mix.length - 1].request
}

async function runMixForDuration(db: Database, runId: number, mix: Array<{ weight: number; request: { target: string; method?: string; headers?: Record<string,string>; body?: string } }>, concurrency: number, durationMs: number, thinkTimeMs?: number) {
  const deadline = Date.now() + durationMs
  async function worker() {
    while (Date.now() < deadline) {
      const req = pickMix(mix)
      const m = (req.method || 'GET').toUpperCase()
      await singleRequest(db, runId, req.target, m, req.body, req.headers)
      if (thinkTimeMs && thinkTimeMs > 0) await new Promise(r => setTimeout(r, thinkTimeMs))
    }
  }
  const workers = Array.from({ length: Math.max(1, concurrency) }, () => worker())
  await Promise.all(workers)
}

const runEmitters = new Map<number, EventEmitter>()
function getRunEmitter(runId: number) {
  let em = runEmitters.get(runId)
  if (!em) { em = new EventEmitter(); runEmitters.set(runId, em) }
  return em
```

```
}

async function executeK6Run(db: Database, runId: number, spec: SimpleSpec, method: string) {
  const tmp = fs.mkdtempSync(path.join(os.tmpdir(), 'perfbench-k6-'))
  const scriptPath = path.join(tmp, `run-${runId}.js`)
  const summaryPath = path.join(tmp, `summary-${runId}.json`)

  const options: any = {}
  if (spec.stages && spec.stages.length) {
     options.stages = spec.stages.map(s => ({ duration: `${Math.max(1, Math.round(s.durationMs/1000))}s`, target: Math.max(1, s.concurrency) }))
  } else if (spec.durationMs && spec.durationMs > 0) {
    options.vus = Math.max(1, spec.concurrency ?? 1)
    options.duration = `${Math.max(1, Math.round(spec.durationMs/1000))}s`
  } else {
    options.vus = Math.max(1, spec.concurrency ?? 1)
    if (spec.requests && spec.requests > 0) options.iterations = spec.requests
  }

  const bodyArg = spec.body ? `, ${JSON.stringify(spec.body)}` : ''
  const script = `import http from 'k6/http';\nexport const options = ${JSON.stringify(options)};\nexport default function () {\n  const res = http.request('${method}', '${spec.target}'${bodyArg});\n}`
  fs.writeFileSync(scriptPath, script)

  await new Promise<void>((resolve, reject) => {
    const proc = spawn('k6', ['run', '--summary-export', summaryPath, scriptPath], { stdio: 'inherit' })
    proc.on('error', reject)
    proc.on('exit', (code) => code === 0 ? resolve() : reject(new Error(`k6 exit code ${code}`)))
  }).catch(err => {
    const notes = JSON.stringify({ engine: 'k6', error: String(err) })
     db.prepare('update runs set status = ?, finishedAt = datetime(\'now\'), notes = ? where id = ?').run('failed', notes, runId)
  })

  try {
    if (fs.existsSync(summaryPath)) {
      const content = fs.readFileSync(summaryPath, 'utf-8')
```

```
    db.prepare('update runs set notes = ? where id = ?').run(JSON.stringify({ engine: 'k6', summary: JSON.parse(content) }), runId)
   }
  } catch {}
}
```