

**МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
 НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ   
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу  
Кафедра системного проектування**

**Звіт**

**про виконання практичної роботи №5  
з дисципліни «Паралельні обчислення»**

Виконав:  
студент III курсу, групи ДА-22  
Жадько Микита Сергійович

Прийняв:

асистент Яременко В. С.

Київ – 2025

**ПОСТАНОВКА ЗАДАЧІ**

1. Спроектувати веб сервер для хостингу статичного контенту. Можна обирати довільну технологією, вимогою є програмування на транспортному (четвертому) рівні стеку OSI (з допомогою сокетів).
2. Веб-сервер повинен опрацьовувати http запити з багатьох клієнтів. Реалізуємо GET відповідно до протоколу HTTP 1.1. RFC9112 RFC9110 Root-запит веде на index.html (приклади запитів “GET http://localhost:8080”, “GET http://localhost:8080/second\_page.html”).
3. Захостити щонайменше 2 веб-сторінки (текст файлів наведено в Додатку 1, або можете додати власні). При цьому повинні оброблятися неіснуючі сторінки. (404 відповідь)
4. Провести навантажувальне тестування з допомогою сервісів jMeter або Locust. Визначити точку відмови в кількості паралельних клієнтів.
5. Зробити висновки, припустити яким чином можна зробити сервер більш ефективним і підвищити його здатність тримати навантаження.
6. (\*) Додаткове завдання на плюс 1 бал – оптимізувати веб-сервер, зробивши його неблокуючим.

**ХІД РОБОТИ**

Спочатку створимо сам код серверу. Наведу його нижче:

#include <winsock2.h>  
#include <ws2tcpip.h>  
#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <string>  
#include <thread>  
  
#define PORT 7890  
#define ROOT\_DIR "static"  
  
using namespace std;  
  
string readFile(const string& path) {  
 ifstream file(path, ios::*binary*);  
 if (!file.is\_open()) return "";  
 ostringstream ss;  
 ss << file.rdbuf();  
 return ss.str();  
}  
  
void sendResponse(SOCKET client, const string& status, const string& content) {  
 ostringstream response;  
 response << "HTTP/1.1 " << status << "\r\n";  
 response << "Content-Length: " << content.size() << "\r\n";  
 response << "Content-Type: text/html\r\n\r\n";  
 response << content;  
 send(client, response.str().c\_str(), response.str().length(), 0);  
}  
  
void handleClient(SOCKET clientSocket) {  
 char buffer[4096];  
 int received = recv(clientSocket, buffer, sizeof(buffer) - 1, 0);  
 if (received <= 0) {  
 closesocket(clientSocket);  
 return;  
 }  
 buffer[received] = '\0';  
  
 istringstream request(buffer);  
 string method, path, version;  
 request >> method >> path >> version;  
  
 if (method != "GET") {  
 string body = "Method is not \"GET\"";  
 sendResponse(clientSocket, body, body);  
 closesocket(clientSocket);  
 return;  
 }  
  
 if (path == "/") path = "/index.html";  
  
 string filePath = string(ROOT\_DIR) + path;  
  
 string content = readFile(filePath);  
 if (content.empty()) {  
 string body = "<html><body><h1>404 Not Found</h1></body></html>";  
 sendResponse(clientSocket, "404 Not Found", body);  
 } else {  
 sendResponse(clientSocket, "200 OK", content);  
 }  
  
 closesocket(clientSocket);  
}  
  
int main() {  
 WSADATA wsaData;  
 if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {  
 cerr << "WSAStartup failed." << endl;  
 return 1;  
 }  
  
 SOCKET serverSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);  
 if (serverSocket == INVALID\_SOCKET) {  
 cerr << "Socket creation failed." << endl;  
 WSACleanup();  
 return 1;  
 }  
  
 sockaddr\_in serverAddr;  
 serverAddr.sin\_family = AF\_INET;  
 serverAddr.sin\_addr.s\_addr = INADDR\_ANY;  
 serverAddr.sin\_port = htons(PORT);  
  
 if (bind(serverSocket, (SOCKADDR\*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR) {  
 cerr << "Bind failed." << endl;  
 closesocket(serverSocket);  
 WSACleanup();  
 return 1;  
 }  
  
 if (listen(serverSocket, SOMAXCONN) == SOCKET\_ERROR) {  
 cerr << "Listen failed." << endl;  
 closesocket(serverSocket);  
 WSACleanup();  
 return 1;  
 }  
  
 cout << "Server started at http://localhost:" << PORT << endl;  
  
 while (true) {  
 SOCKET clientSocket = accept(serverSocket, NULL, NULL);  
 if (clientSocket == INVALID\_SOCKET) {  
 cerr << "Accept failed." << endl;  
 continue;  
 }  
 thread(handleClient, clientSocket).detach();  
 }  
  
 closesocket(serverSocket);  
 WSACleanup();  
 return 0;  
}

Далі також створимо дві веб сторінки. А саме

index.html:

<!DOCTYPE html>  
<html>  
<head><title>Page 1</title></head>  
<body>  
<h1>Welcome to Page 1</h1>  
<p>Click <a href="page2.html">here</a> to go to Page 2.</p>  
</body>  
</html>

page2.html:

<!DOCTYPE html>  
<html>  
<head><title>Page 2</title></head>  
<body>  
<h1>Welcome to Page 2</h1>  
<p>Click <a href="index.html">here</a> to go back to Page 1.</p>  
</body>  
</html>

Також створимо файл для тестування locustfile.py:

from locust import HttpUser, task

class StaticWebUser(HttpUser):

@task

def load\_index(self):

self.client.get("/index.html")

@task

def load\_page2(self):

self.client.get("/page2.html")

@task

def load\_404(self):

self.client.get("/nonexistent.html")

Вийде приблизно така структура файлів (програмую я у CLion):

* Server (головна папка)
  + cmake-build-debug
    - static
      * index.html
      * page2.html
    - locustfile.py
    - …
    - Server.exe
  + …
  + main.cpp

Далі, для тестування, треба встановити Locust. Для цього завантажимо з офіційного сайту Python та налаштуємо його.

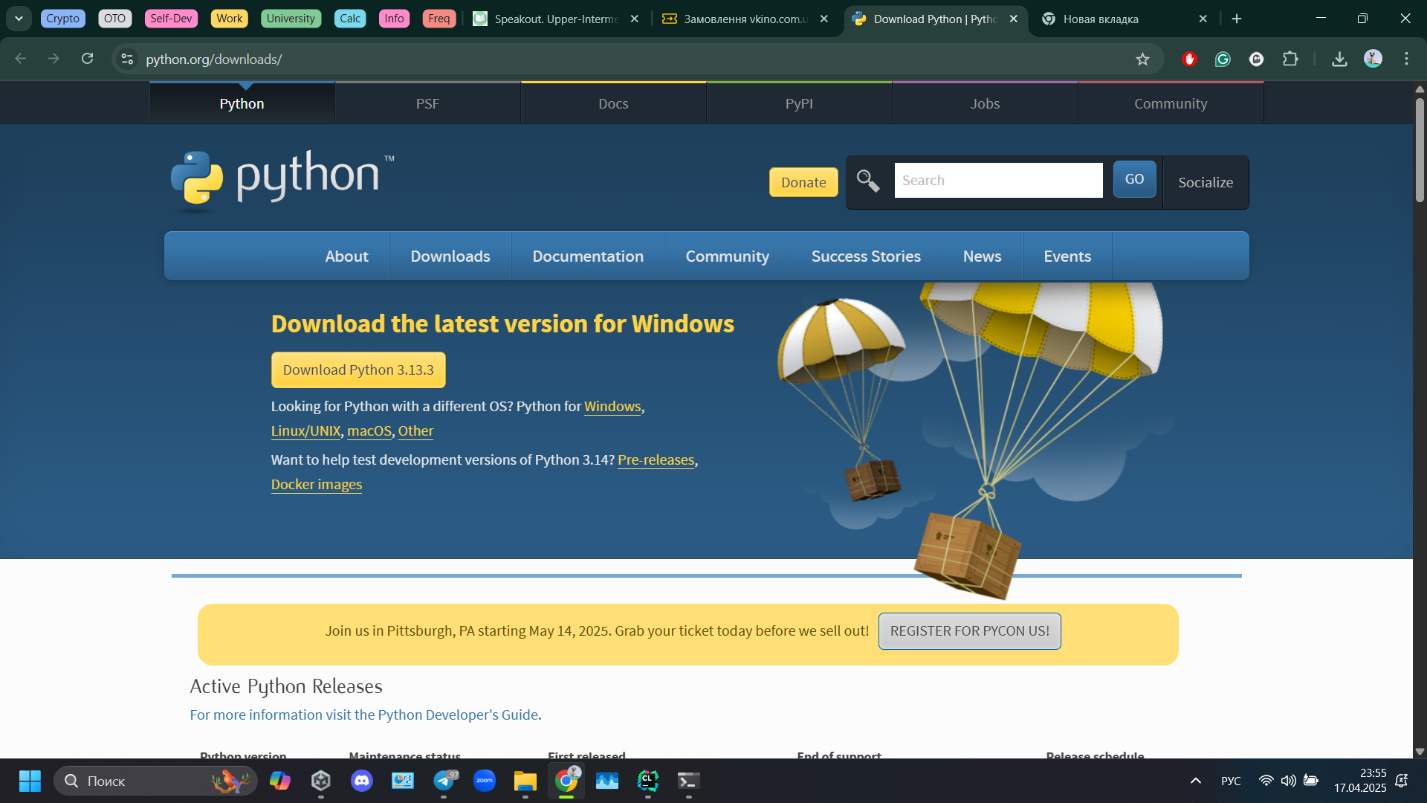


Рисунок 1 — Завантажуємо Python.

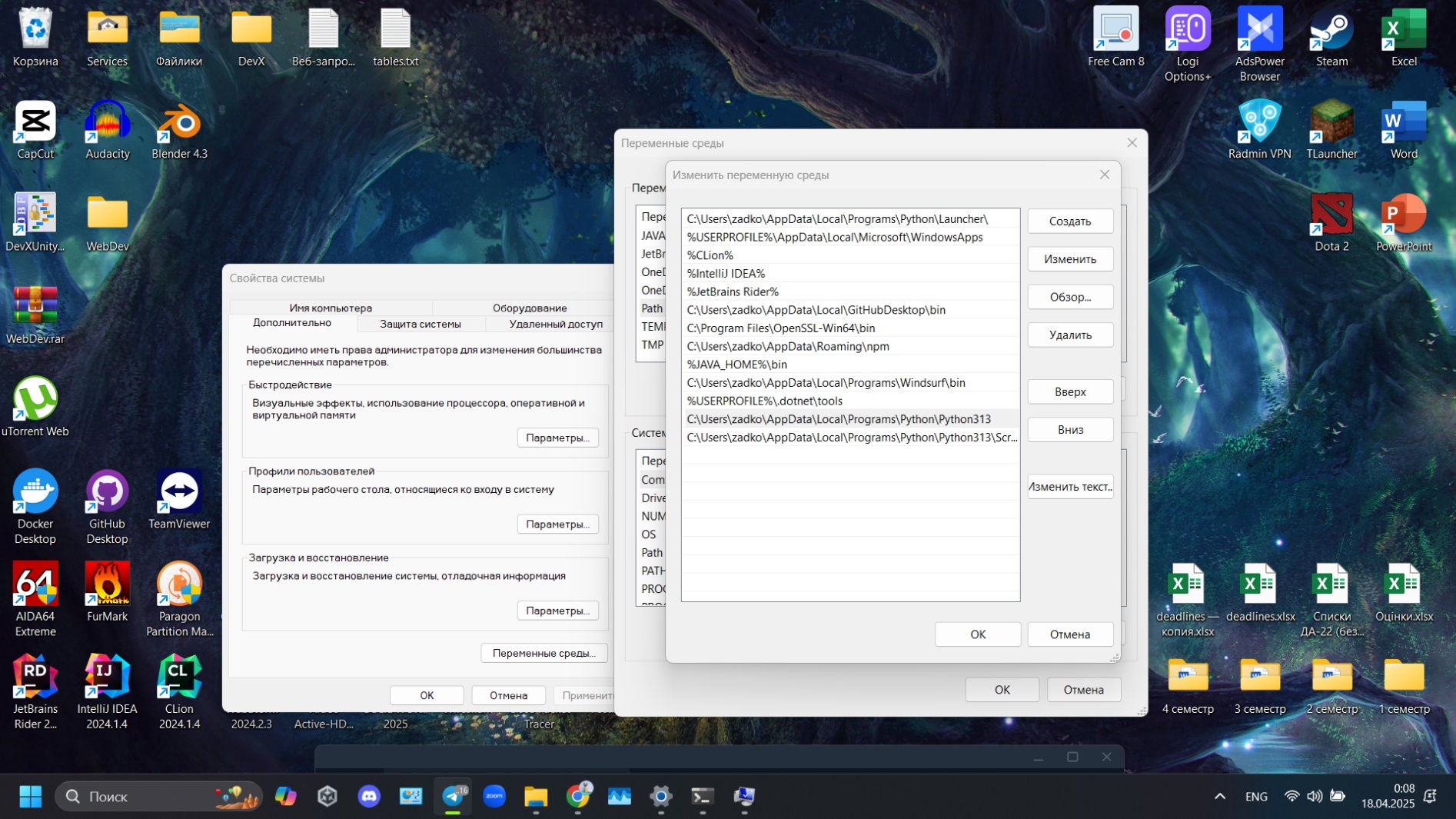


Рисунок 2 — Налаштовуємо змінні оточення.

Далі треба встановити сам locust

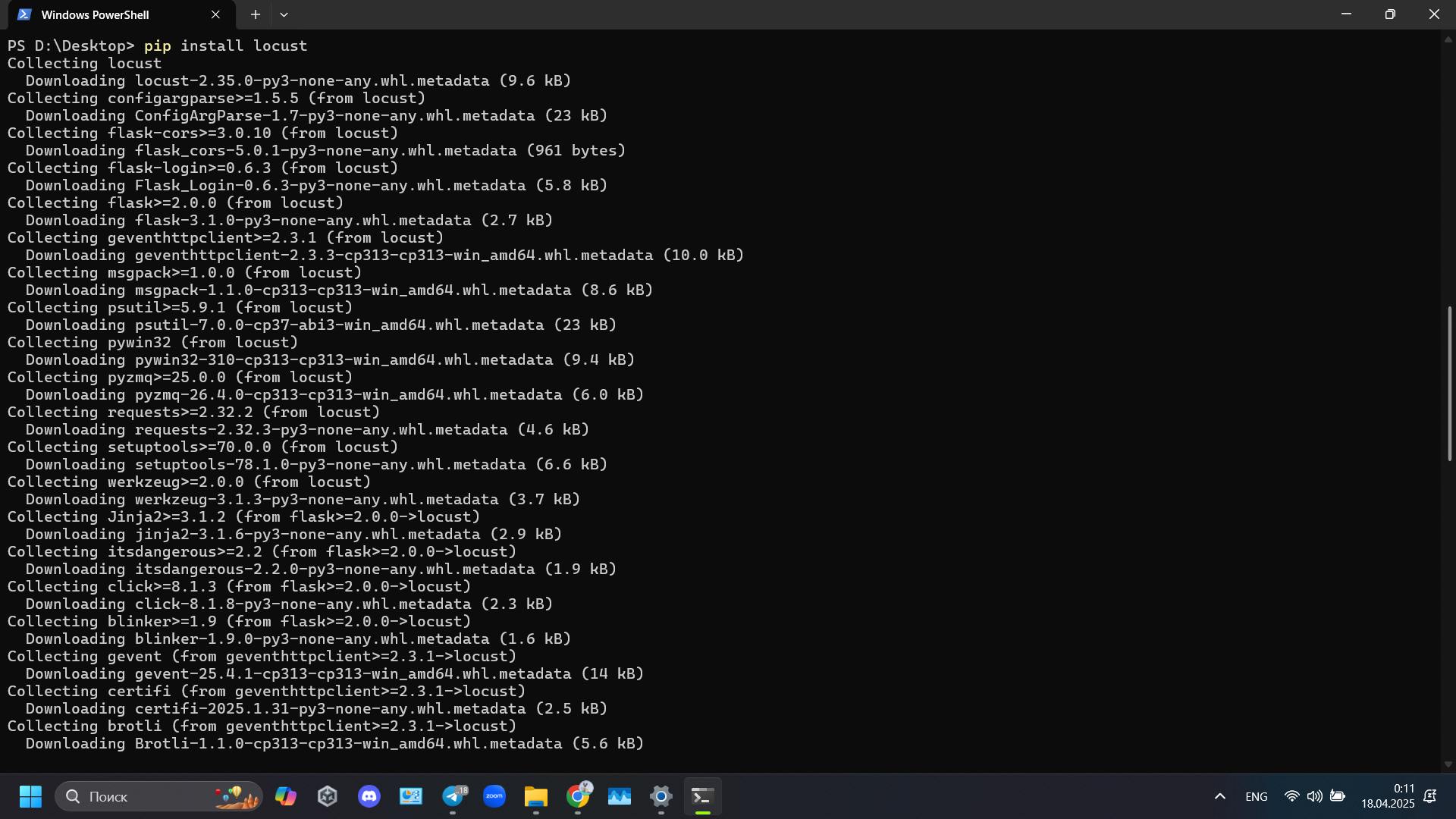


Рисунок 3 — Встановлення locust.

Далі треба запустити наш сервер

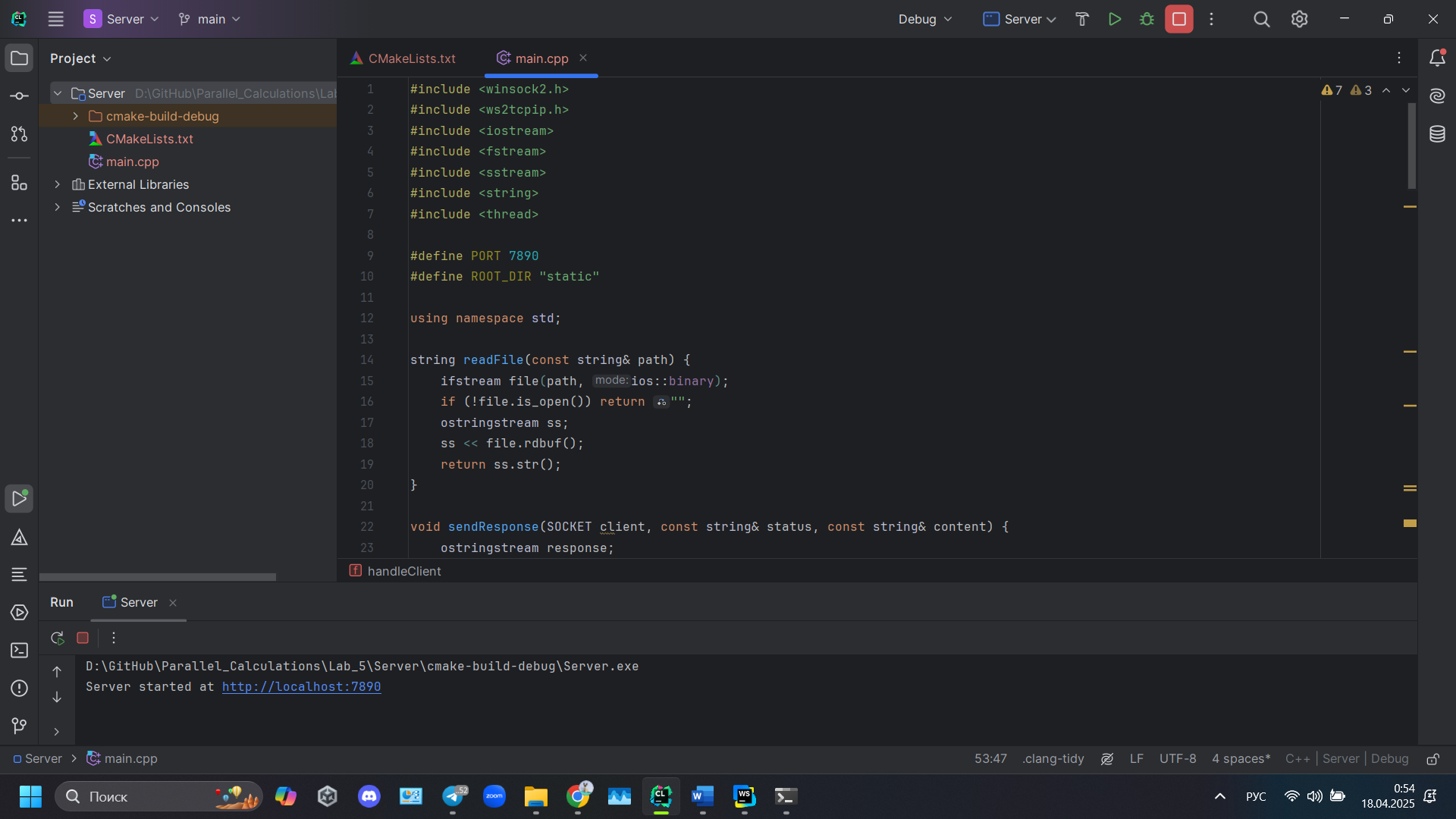


Рисунок 4 — Запуск коду сервера.

І також запустити сам locust

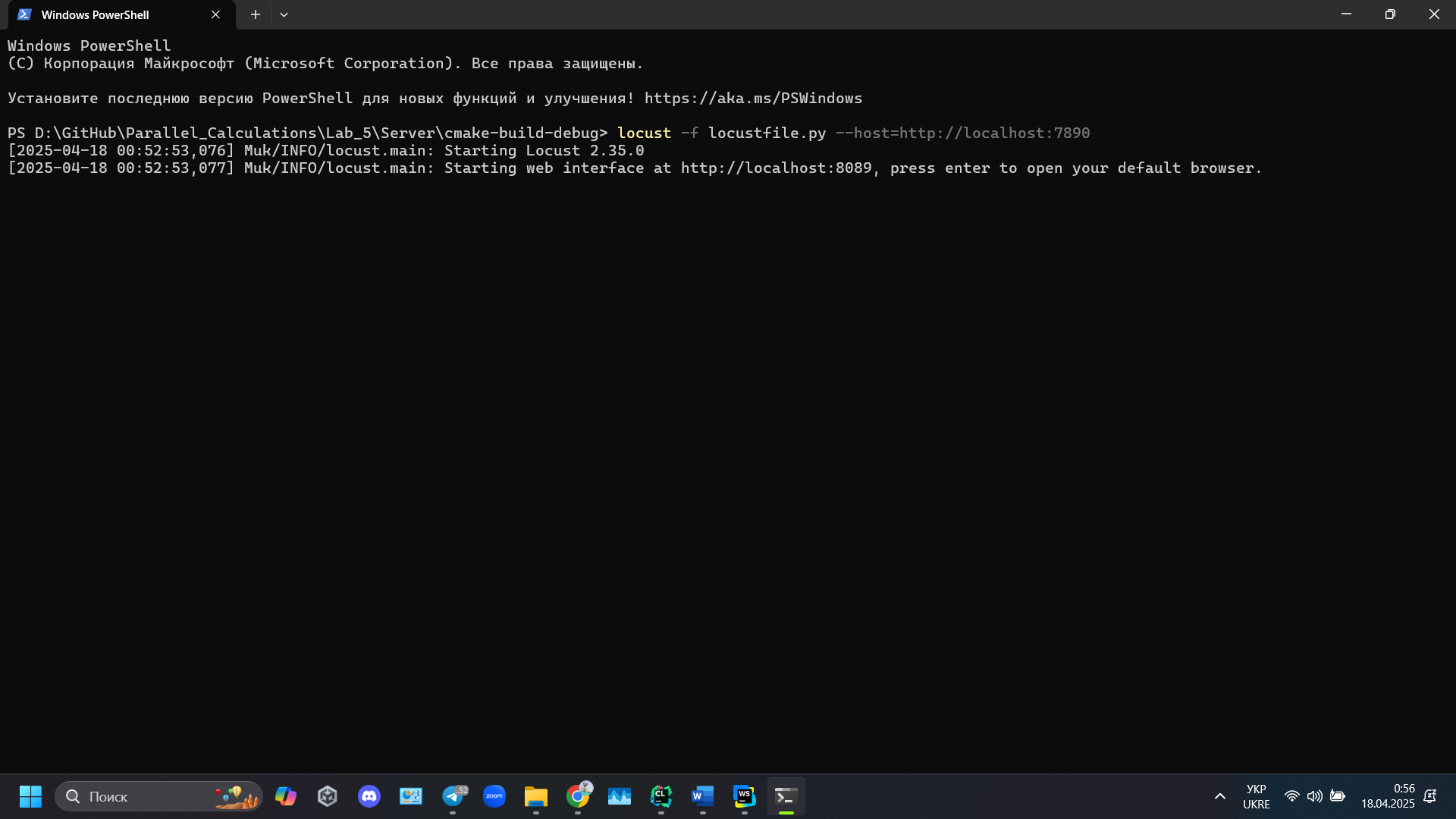


Рисунок 5 — Запускаємо locust з директорії, в якій у нас знаходиться файл locustfile.py, вказавши адресу з портом, який ми вказали у коді сервера.

Тепер перейдемо за посиланням <http://localhost:8089/>, де нас зустріне приємний інтерфейс

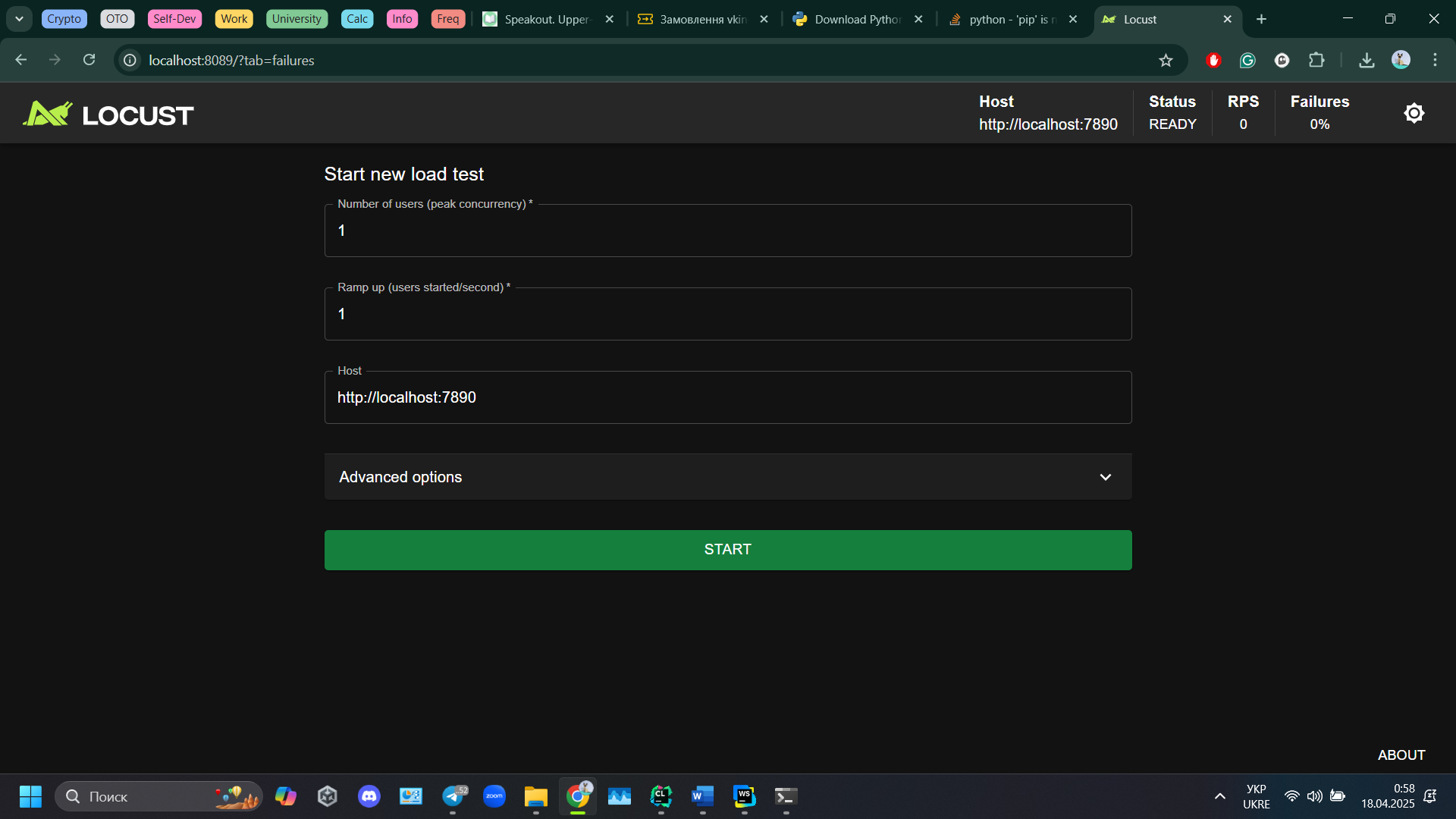


Рисунок 6 — Інтерфейс locust.

Для початку запустимо тести з 1 юзером на секунду та подивимося на правильність роботи програми.

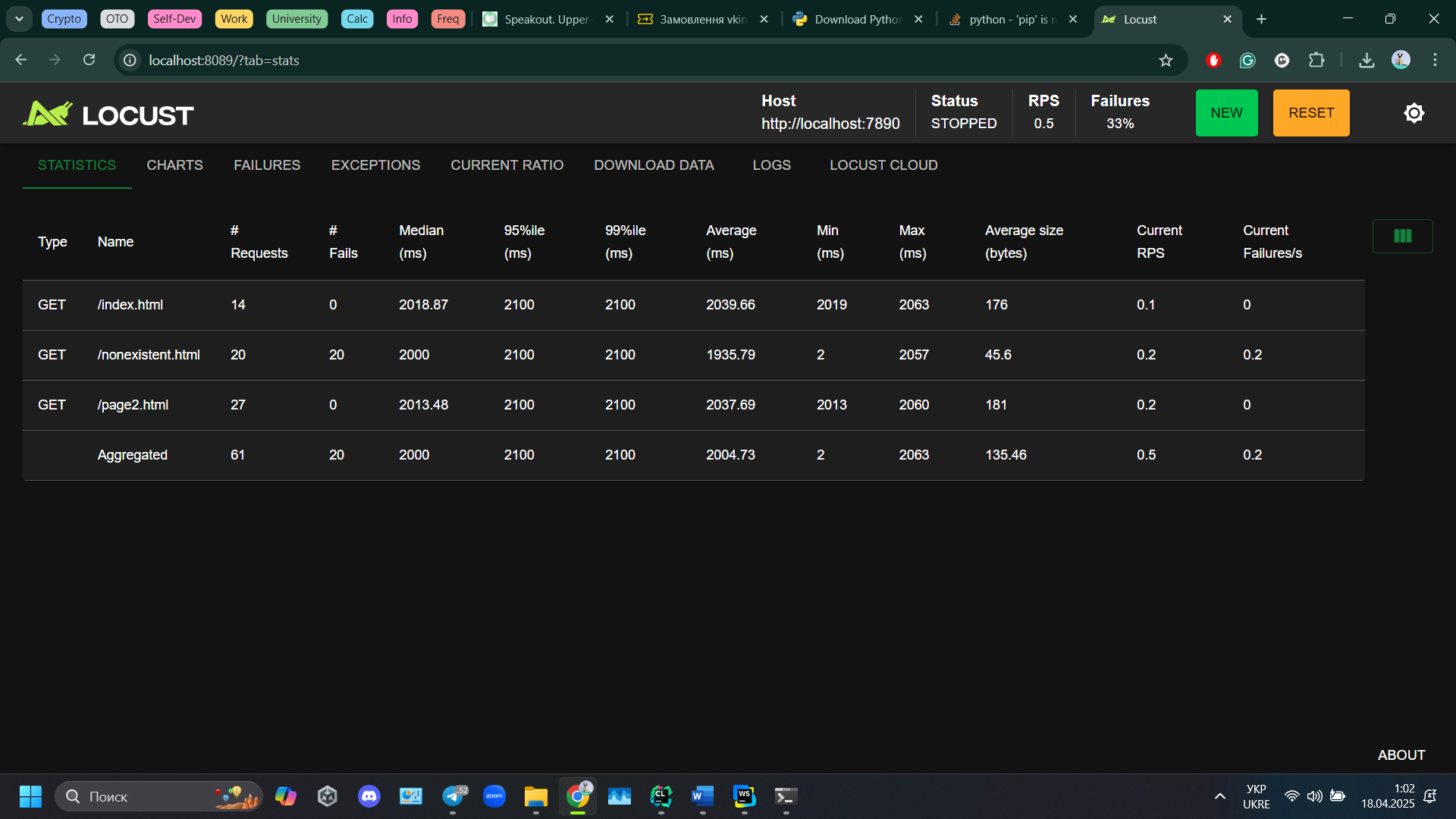


Рисунок 7 — Результати роботи програми.

Як бачимо, хороші сайти не відвалилися, а погані сайти не відповідають.

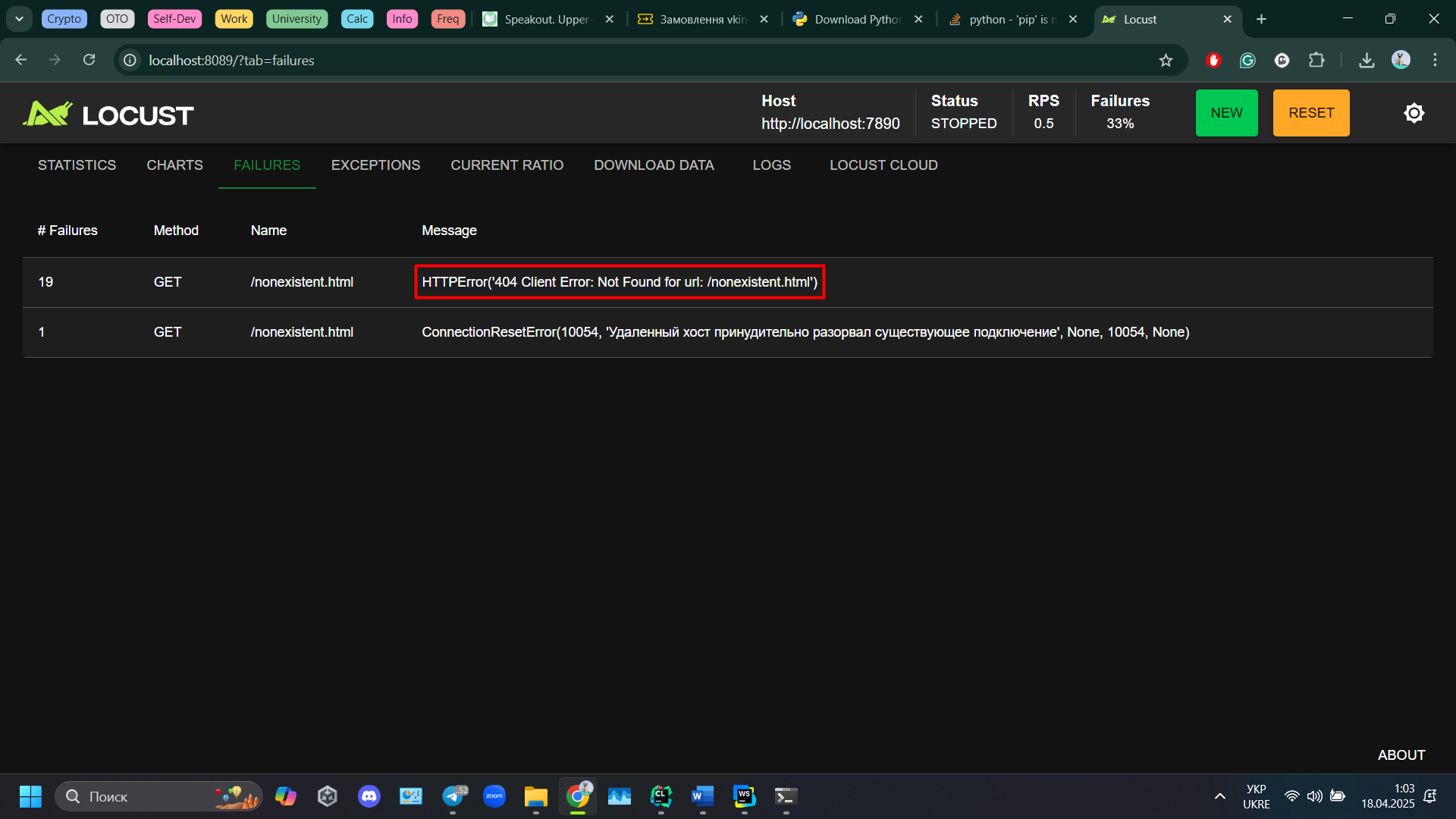


Рисунок 8 — Текст помилки для не існуючого сайту.

Як бачимо, не існуючий сайт повертає помилку 404, що прописана в коді. Тепер приберемо цей сайт та протестимо два основних. Для цього змінимо locustfile.py та перезапустимо locust

Візьмемо одразу 1000 користувачів одночасно.

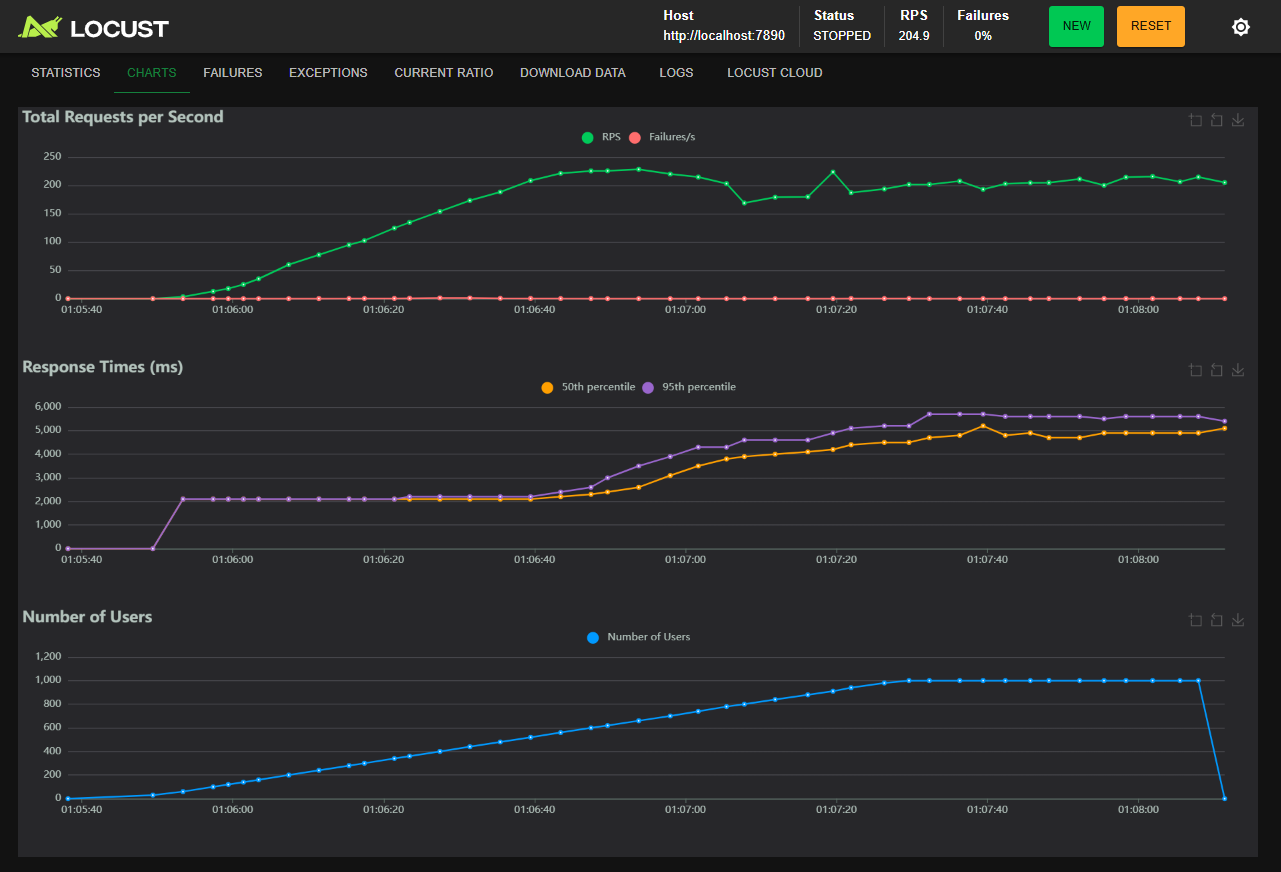


Рисунок 9 — Графіки для 1000 користувачів пікового навантаження.

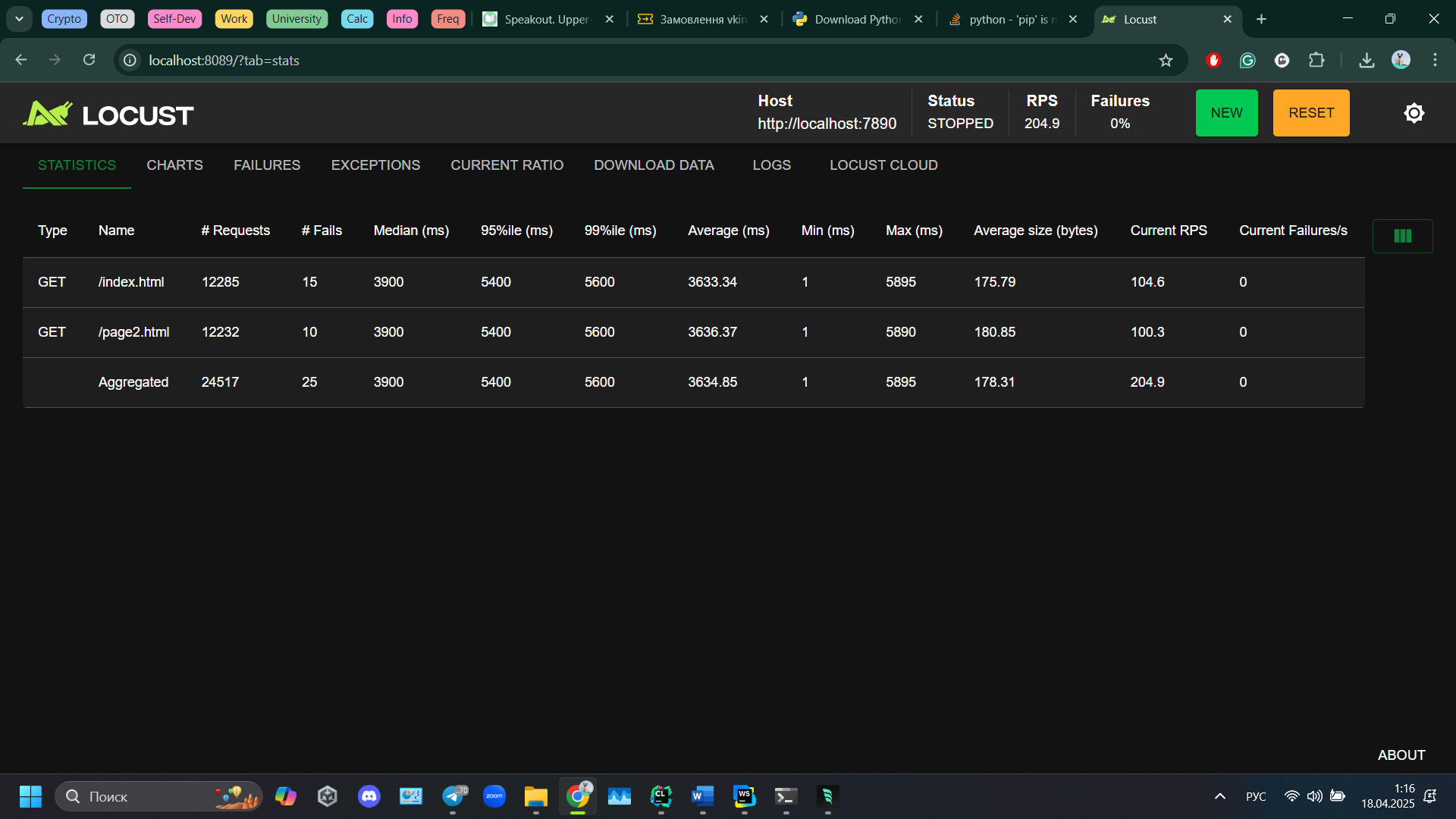


Рисунок 10 — Вийшло приблизно 200 запитів в секунду, але не всі успішні.

Що цікаво, якщо запустити 5000 юзерів одночасно, то ми отримаємо приблизно ті ж запити в секунду, але жодної помилки. Натомість час запиту збільшиться в середньому у 5 разів.

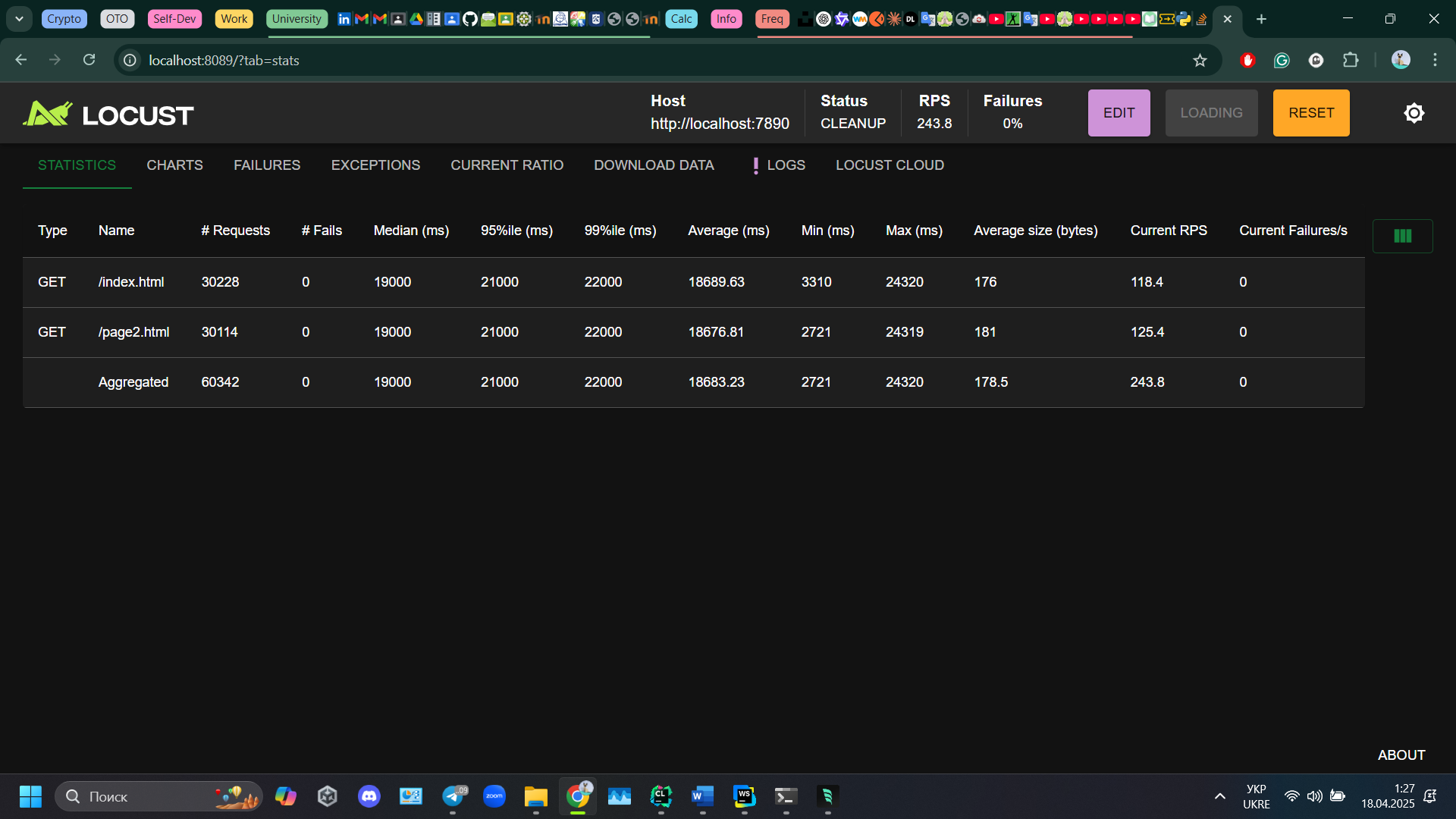


Рисунок 11 — 5000 юзерів одночасно.

**ВИСНОВКИ**

У результаті виконання лабораторної роботи було розроблено простий HTTP-сервер для обслуговування статичного контенту на основі сокетів з використанням WinSock API. Сервер реалізує обробку HTTP GET-запитів відповідно до специфікацій протоколу HTTP/1.1 та підтримує одночасну взаємодію з великою кількістю клієнтів шляхом запуску окремих потоків для кожного з’єднання.

Протягом розробки серверу було реалізовано маршрутизацію запитів, обробку коректних та некоректних шляхів, а також генерацію відповіді 404 для неіснуючих сторінок. Було створено дві HTML-сторінки для демонстрації роботи та навігації між ними. Особливу увагу було приділено роботі з файлами, кодуванню відповідей та обробці помилок з’єднання.

Для оцінки продуктивності проведено навантажувальне тестування за допомогою інструменту Locust. У процесі тестування було визначено, що сервер здатний стабільно обслуговувати понад 200 запитів на секунду, однак із зростанням навантаження спостерігається збільшення часу відповіді. При максимальному навантаженні у 5000 паралельних клієнтів сервер не падає, але середній час відповіді збільшується у кілька разів, що свідчить про обмеження його масштабованості в поточній реалізації.

Для подальшого покращення продуктивності можливо реалізувати неблокуючий I/O або використовувати пул потоків замість створення нового потоку на кожне з’єднання. Це дозволить зменшити накладні витрати на створення потоків та покращити масштабованість рішення.

Загалом, розроблений застосунок демонструє принципи побудови простого HTTP-сервера на транспортному рівні OSI-моделі та дозволяє ефективно обробляти запити від великої кількості клієнтів.

Також [посилання на репозиторій](https://github.com/nikk0308/Parallel_Calculations).