МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ПАРАЛЕЛЬНОГО ВЕБСЕРВЕРУ ДЛЯ ЧИСЕЛЬНОГО МОДЕЛЮВАННЯ»**

Виконав: студент 4 курсу, групи 6.1211-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.Р. Карпенко
(ініціали та прізвище)
Керівник декан математичного факультету, професор, д.т.н. Гоменюк С. І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Карпенку Валерію Романовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка паралельного вебсерверу для чисельного моделювання

керівник роботи Гоменюк Сергій Іванович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі, аналіз предметної області.
2. Проєктування.
3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	20.01.2025	
2.	Збір вихідних даних.	10.02.2025	
3.	Обробка методичних та теоретичних	10.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	21.04.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	18.06.2025	

Студент _____ В.Р. Карпенко
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.І. Гоменюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка паралельного вебсерверу для чисельного моделювання»: 62 с., 31 рис., 2 табл., 21 джерело, 5 додатків.

AGENT-BASED MODELING, ANGULAR, CELERY, DOCKER, FASTAPI, NUMERICAL MODELING, PARALLEL COMPUTING, REDIS, REST API, UML.

Об’єкт дослідження – процес чисельного моделювання.

Мета роботи – розробити паралельний вебсервер для чисельного моделювання.

Методи дослідження – моделювання, проєктування, програмний, аналітичний.

У роботі розглянуто проєктування та реалізацію вебзастосунку для чисельного моделювання з використанням Angular, FastAPI, Celery і Redis. Застосовано UML-діаграми варіантів використання, діяльності, послідовності та розгортання. Система дозволяє запускати симуляцію з параметрами, заданими користувачем, та переглядати результати у вигляді анімації, графіків і таблиць.

Для забезпечення високої продуктивності симуляцій реалізовано паралельне виконання задач за допомогою Celery у фоновому режимі. Архітектура системи побудована на базі контейнерів Docker. Фронтенд реалізовано на Angular з адаптивним інтерфейсом, який підтримує візуалізацію сітки агентів у Canvas, побудову графіків станів та покрокове керування симуляцією. Передбачено unit- та інтеграційне тестування, а також аналіз часу виконання при різній кількості ядер.

Таким чином, розроблений вебзастосунок забезпечує ефективне паралельне чисельне моделювання з візуальною інтерпретацією динаміки агентної системи. Система придатна до масштабування, налаштування, і може використовуватись у дослідницьких або навчальних цілях.

SUMMARY

Bachelor’s qualifying paper «Development of a Parallel Web Server for Numerical Modeling»: 62 pages, 31 figures, 2 tables, 21 references, 5 supplements.

AGENT-BASED MODELING, ANGULAR, CELERY, DOCKER, FASTAPI, NUMERICAL MODELING, PARALLEL COMPUTING, REDIS, REST API, UML.

The object of the study is the process of numerical modeling.

The aim of the study is to develop a parallel web server for numerical modeling.

The methods of research are modeling, design, programming, analytical.

The paper discusses the design and implementation of a web application for numerical simulation using Angular, FastAPI, Celery, and Redis. UML diagrams of use cases, activities, sequence, and deployment are used. The system allows you to run simulations with user-defined parameters and view the results in the form of animations, graphs, and tables.

To ensure high performance of simulations, parallel execution of tasks using Celery in the background is implemented. The system architecture is based on Docker containers. The frontend is implemented in Angular with an adaptive interface that supports visualization of the agent grid in Canvas, construction of state graphs, and step-by-step simulation control. Unit and integration testing, as well as analysis of execution time with different number of cores are provided.

Thus, the developed web application provides efficient parallel numerical modeling with visual interpretation of the agent system dynamics. The system is scalable, customizable, and can be used for research or educational purposes.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання 10
1.1 Терміни та визначення 10
1.1.1 Загальні терміни 10
1.1.2 Технічні терміни 10
1.2 Функціональні вимоги 11
1.2.1 Призначення і цілі створення системи 11
1.2.2 Загальні функціональні можливості системи 11
1.3 Нефункціональні вимоги 12
1.4 Опис предметної області 12
1.5 Опис системи 13
1.6 Методи та алгоритми чисельного моделювання 13
1.7 Сучасні засоби реалізації паралельних чисельних симуляцій 14
1.8 Аналіз та огляд аналогів 15
2 Проєктування 18
2.1 Використання UML під час розробки системи 18
2.2 Діаграма варіантів використання 19
2.2.1 Опис варіантів використання 21
2.3 Діаграма діяльності 28
2.4 Діаграма послідовності 30
2.5 Діаграма розгортання 31
3 Реалізація та тестування 33
3.1 Опис інструментів розробки 33
3.2 Основні класи системи 33

3.3 Встановлення та налаштування оточення 35
3.4 Опис API 38
3.5 Тестування проєкту 39
3.6 Керівництво користувача 40
3.6.1 Рівень підготовки користувача 40
3.6.2 Відкриття застосунку та ознайомлення з параметрами 40
3.6.3 Запуск симуляції 42
3.6.4 Перегляд результатів симуляції 42
3.6.5 Перегляд по кроках 44
3.6.6 Повторний запуск симуляції 46
3.7 Порівняння часу симуляції при різній кількості ядер 47
Висновки 49
Перелік посилань 50
Додаток А SimulationComponent 52
Додаток Б SimulationService 58
Додаток В AgentSimulationModel 59
Додаток Г Docker compose 61
Додаток Д Angular-тести 62

ВСТУП

Чисельне моделювання стало важливим інструментом у дослідженні складних динамічних процесів, таких як поширення інфекцій, поведінка соціальних груп, міграційні процеси тощо. Одним з найпоширеніших підходів у цьому контексті є моделювання на основі агентів, яке дозволяє відтворити поведінку великої кількості незалежних одиниць у певному середовищі.

З розвитком вебтехнологій виникла потреба у створенні зручних, доступних і масштабованих систем, які б дозволяли проводити чисельне моделювання без необхідності встановлення спеціалізованого програмного забезпечення. Водночас, зростаючий обсяг моделювань потребує оптимізації продуктивності, що забезпечується завдяки використанню паралельних обчислень.

Існуючі інструменти для моделювання часто орієнтовані на дослідників і вимагають програмних навичок, не мають інтерактивного інтерфейсу або не забезпечують належної швидкодії в реальному часі. Це створює труднощі для їх застосування в навчальних, аналітичних або демонстраційних цілях.

Виходячи з цього, виникла потреба у створенні сучасного вебзастосунку, який дозволяє запускати симуляції агентних систем з можливістю задавання параметрів, перегляду візуалізації, статистики та графіків – і все це у зручному інтерактивному середовищі з підтримкою паралельних обчислень.

Актуальність дослідження зумовлена необхідністю створення масштабованого веборієнтованого інструменту для чисельного моделювання з підтримкою візуалізації результатів, що може бути використаний як у дослідженнях, так і в освітньому процесі.

З огляду на це, можна виділити наступні цілі і задачі дослідження.

Мета: розробити паралельний вебсервер для чисельного моделювання на прикладі агентних систем.

Досягнення поставленої мети передбачає реалізацію наступних завдань:

– сформувати вимоги до вебсерверу;
– спроєктувати та побудувати архітектуру вебсерверу;
– реалізувати паралельний вебсервер для чисельного моделювання на прикладі агентних систем;
– протестувати роботу вебсерверу.

Об’єкт дослідження: процес чисельного моделювання.

Предмет дослідження: фреймворки FastAPI, Celery, Angular та розподілене сховище Redis.

Методи дослідження: моделювання, проєктування, програмний, аналітичний.

У першому розділі здійснено аналіз предметної області, сформовано вимоги до системи, описано методи чисельного моделювання, а також розглянуто існуючі аналоги.

У другому розділі побудовано архітектуру системи, надано UML-діаграми, описано основні класи та взаємодію між ними.

У третьому розділі подано реалізацію системи, приклади використання, результати тестування, порівняння продуктивності та керівництво користувача.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Терміни та визначення

1.1.1 Загальні терміни

Система – вебзастосунок для виконання паралельного чисельного моделювання динаміки агентних систем у вигляді симуляцій.

Клієнт – фронтенд частина застосунку, реалізована засобами Angular.

Сервер – бекенд частина системи, побудована на FastAPI із використанням Celery для фонових паралельних задач.

Симуляція – модель, яка імітує поведінку великої кількості агентів із можливістю зараження, одужання, переміщення.

Користувач – особа, яка взаємодіє із застосунком через графічний інтерфейс.

Агент – умовна одиниця симуляції (наприклад, особа), що має стан (здоровий, інфікований, вилікуваний) та правила поведінки.

1.1.2 Технічні терміни

FastAPI – сучасний асинхронний фреймворк для розробки REST API на Python [9].

Celery – система керування чергами задач, яка дозволяє виконувати обчислення у фоновому режимі [4].

Redis – брокер повідомлень для передачі задач між FastAPI та Celery [19].

Angular – TypeScript-фреймворк для побудови реактивних односторінкових застосунків [1].

RxJS – бібліотека для роботи з реактивним програмуванням в Angular [20].

Docker – інструмент для контейнеризації застосунків [8].

Canvas – HTML-елемент для рендеру графіки, який використовується для візуалізації симуляції.

Chart.js – бібліотека для побудови графіків, використовувана у проєкті через ng2-charts [5, 17].

1.2 Функціональні вимоги

1.2.1 Призначення і цілі створення системи

Функціональне призначення системи – реалізувати можливість запуску симуляції агентної системи з параметрами користувача та виведення результатів у зручному інтерфейсі.

Експлуатаційне призначення – система працює у режимі клієнт–сервер, з використанням паралельних обчислень для зменшення часу розрахунку.

Мета створення системи – розробка сучасного вебінтерфейсу для виконання чисельного моделювання з можливістю візуалізації результатів та подальшого аналізу.

1.2.2 Загальні функціональні можливості системи

Система має надавати користувачам наступні можливості:

– запускати симуляцію із вказаними параметрами (розмір сітки, кількість агентів, кроки, ймовірність зараження та час одужання);
– переглядати симуляцію на екрані у вигляді інтерактивної сітки;
– переглядати графік зміни станів (здорові, інфіковані, вилікувані);
– переглядати теплову карту зараження;
– переглядати зведену статистику;
– переглядати кроки симуляції у табличному вигляді;
– покроково переміщатися по кадрах симуляції.

1.3 Нефункціональні вимоги

Інтерфейс користувача. Інтерфейс користувача повинен бути адаптивним та інтуїтивно зрозумілим. Усі дії мають супроводжуватися зворотним зв’язком (наприклад, індикатор завантаження, повідомлення про помилки).

Підтримка браузерів. Система має підтримувати останні стабільні версії Chrome, Firefox, Edge, Safari, а також гарантувати сумісність та доступність як на мобільних, так і на десктопних пристроях.

Вимоги до продуктивності. Сервер має забезпечувати виконання симуляції в межах прийнятного часу (до 5 секунд при середніх значеннях параметрів), що повинно досягатися за допомогою використанню Celery і багатоядерності.

Вимоги до безпеки. Система повинна мати захист від некоректного вводу. Сервер повинен перевіряти валідність даних перед обчисленням. Користувач не повинен мати прямого доступу до системи задач або файлової системи сервера.

1.4 Опис предметної області

Симуляція агентних систем – це тип чисельного моделювання, що передбачає взаємодію множини об’єктів (агентів), які діють згідно заданих правил [11].

У даному випадку реалізується симуляція поширення інфекції на сітці: кожен агент має певний стан і може змінювати його залежно від оточення. Це дозволяє моделювати соціальні процеси, епідемії, мобільність населення [11].

Користувач визначає параметри симуляції, після чого система обчислює її результат у фоновому режимі. Результати подаються у вигляді покадрової візуалізації, графіків та теплових карт.

1.5 Опис системи

Розроблена система складається з двох основних частин:

– клієнтська на Angular, яка відповідає за збір параметрів симуляції, відображення візуалізації, статистики та графіків;
– серверна на FastAPI + Celery, яка відповідає за виконання чисельного моделювання в окремому процесі та повернення результатів.

Фонові обчислення виконуються асинхронно з використанням черги задач, для цього використовується Redis, який виступає як брокер [19]. Система підтримує масштабування за рахунок запуску кількох воркерів, що дає змогу прискорити симуляцію при збільшенні кількості ядер [4].

1.6 Методи та алгоритми чисельного моделювання

Чисельне моделювання – це розділ прикладної математики, який займається створенням алгоритмів для розв’язання задач, що не мають аналітичного розв’язку або мають занадто складну форму. Однією з найбільш поширених прикладних технік є агентне моделювання (agent-based modeling, ABM), яке дозволяє аналізувати поведінку великої кількості автономних одиниць (агентів), що взаємодіють у спільному середовищі [11, 12].

У даному проєкті використовується дискретна модель на основі клітинного автомата. Простір моделі представлено у вигляді двовимірної сітки, де кожна клітинка відповідає одному агенту [11].

Кожен агент має один із трьох станів:

– 0 – здоровий;
– 1 – інфікований;
– 2 – вилікуваний.

Поведінка агентів задається за допомогою локальних правил переходу між станами. На кожному кроці симуляції кожен інфікований агент має потенціал

заражати сусідні клітинки з певною ймовірністю, а після фіксованої кількості кроків – переходить у стан «вилікуваний». Таким чином, поведінка системи формується знизу-вгору (bottom-up), а макрорезультат є наслідком мікровзаємодій [11].

Цей підхід є ефективним для моделювання епідемій, соціальних процесів, мобільності населення, міграційних потоків, динаміки ринку та інших складних систем. Алгоритм реалізовано у вигляді багаторазового циклу, в якому обчислюється новий стан усієї сітки на основі поточного стану та параметрів [11].

1.7 Сучасні засоби реалізації паралельних чисельних симуляцій

Паралельне чисельне моделювання є актуальним напрямом, оскільки значна кількість моделей потребує обчислення великих обсягів даних у короткі проміжки часу [12, 15]. Сучасні підходи до реалізації таких симуляцій спираються на розпаралелювання процесів, тобто поділ задачі на менші частини, які виконуються одночасно на декількох ядрах або вузлах.

Найпоширеніші інструменти, які застосовуються в системах на базі Python:

- Celery: фреймворк для асинхронного виконання задач через черги повідомлень – дає змогу запускати задачі у фоновому режимі, відокремлено від основного потоку запитів; для черги задач найчастіше використовують Redis як брокер [4];
- Dask: бібліотека, яка дозволяє виконувати паралельні обчислення на масивах, DataFrame, потоках, застосовуючи розподілений граф обчислень [7];
- Joblib, Multiprocessing, ThreadPoolExecutor: вбудовані в Python інструменти для локального розпаралелювання;
- Ray: сучасна масштабована платформа для побудови паралельних і розподілених Python-додатків [18];

– MPI (Message Passing Interface): класична технологія для високопродуктивних обчислень на кластерах (але не застосовується у вебпроєктах).

У розроблюваній системі реалізовано паралельну обробку симуляцій за допомогою Celery, що дозволяє:

– запускати обчислення незалежно від основного потоку запитів;
– підвищити швидкодію при великій кількості агентів або кроків;
– гнучко контролювати кількість ядер/воркерів через змінні оточення (наприклад, CELERYD_CONCURRENCY);
– масштабувати розрахунки горизонтально (через запуск додаткових контейнерів з воркерами у Docker).

Таким чином, обраний підхід забезпечує ефективну, масштабовану та стабільну архітектуру для чисельного моделювання в реальному часі, з можливістю адаптації під різні обчислювальні потужності.

1.8 Аналіз та огляд аналогів

Серед доступних рішень, схожих за призначенням, можна відзначити: NetLogo, Mesa та AnyLogic.

Розглянемо ці рішення детальніше та порівняємо з розроблюваною системою.

NetLogo – настільне середовище для моделювання агентних систем, розроблене для використання в освіті та наукових дослідженнях. Воно має зручний інтерфейс та дозволяє легко створювати моделі без глибоких знань програмування. До відносних недоліків можна віднести відсутнійсть вебінтерфейсу та погану масштабованість, яка необхідна для паралельних обчислень [16].

Mesa – це Python-фреймворк для створення агентних моделей. Він надає базові інструменти для побудови логіки агентів, простору середовища та

візуалізації. Mesa орієнтований на науковців і має обмежену підтримку паралелізації. Візуалізація реалізується через стандартні бібліотеки або Flask. Система потребує програмування та не має готового GUI для кінцевого користувача [13].

AnyLogic – комерційне програмне забезпечення для моделювання бізнес-процесів, соціальних систем та агентних моделей. Воно підтримує масштабованість, паралельність, інтеграцію зі статистичними методами та базами даних. AnyLogic пропонує потужний візуальний редактор, але вимагає ліцензії, що обмежує його доступність у навчальних чи відкритих проєктах [2].

На основі опису аналогів, відобразимо порівняння з розроблюваною системою (табл. 1.1).

Таблиця 1.1 – Порівняння аналогів

Назва	**Тип**	**Паралельність**	**Візуалізація**
NetLogo	Desktop/Java	Ні	Так
Mesa	Python-фреймворк	Обмежена	Через бібліотеки
AnyLogic	Комерційне	Так	Так
Розроблювана система	Web (Angular + Python)	Так (через Celery)	Так (Canvas, графіки, heatmap)

Порівнюючи розроблювану систему з існуючими аналогами, можна виділити такі її переваги:

– повна автономність користувача (без встановлення програм);
– використання сучасних web-технологій;
– гнучке масштабування завдяки Docker + Celery;
– адаптація для демонстрацій та навчання.

Підводячі підсумок, можна сказати, що розроблювана система має переваги завдяки веборієнтованому рішення, яке поєднує FastAPI, Celery та

Angular для реалізації масштабованої, інтерактивної симуляції агентних систем. Завдяки використанню Docker та паралельного обчислення на бекенді – дозволяє гнучко запускати обчислення з різними параметрами. Користувач отримує не лише візуалізацію, а й графіки, таблиці та статистику, що робить її ефективною для аналізу поведінки моделей.

2 ПРОЄКТУВАННЯ

2.1 Використання UML під час розробки системи

Уніфікована мова моделювання (UML) – це стандартизована мова моделювання, яка широко використовується у сфері розробки програмного забезпечення. Вона забезпечує стандартний спосіб візуалізації структури та поведінки системи, що значно полегшує процес аналізу, проєктування, впровадження та підтримки програмних продуктів [14].

Основною перевагою UML є здатність передавати складні концепції за допомогою діаграм, які зрозумілі як розробникам, так і іншим зацікавленим сторонам, включаючи аналітиків, тестувальників і клієнтів. На початкових стадіях розробки системи UML спрощує процес опису вимог до системи, ідентифікацію основних компонентів, опис їхньої взаємодії та потоків даних. Наприклад, діаграма варіантів використання відповідає за опис функціональності системи з точки зору користувача, а не розробника [10].

На етапі проєктування використовуються різноманітні діаграми UML, найбільш використовуваними з них є діаграми класів, діаграми діяльності та діаграми послідовності. Вони дозволяють будувати складні моделі, які відображають внутрішню архітектуру системи, виділяючи її компоненти, атрибути, методи та динаміку, що керує їхньою взаємодією. Цей підхід сприяє глибшому розумінню архітектури майбутньої програми, полегшуючи тим самим виявлення потенційних помилок кодування ще до початку програмної діяльності [10].

Також UML відіграє ключову роль у документуванні реалізованої системи. Його використання при розробці моделей програмного забезпечення полегшує підтримку та вдосконалення цих систем. Такий підхід дозволяє новим членам команди швидко зрозуміти структуру і логіку системи, тим самим підвищуючи загальну ефективність і продуктивність команди [10, 14].

Отже, з вищеперерахованого, можна зробити висновок, що UML є важливим інструментом, який підвищує ефективність розробки програмних систем. Він сприяє кращому плануванню, ефективній командній комунікації та ефективнішому управлінню складністю проєктів. Його використання дозволяє розробникам створювати більш надійні, зрозумілі та масштабовані програмні додатки.

2.2 Діаграма варіантів використання

Діаграма варіантів використання є ключовим інструментом UML, який використовується для моделювання функціональної поведінки системи з точки зору користувача. Вона призначена для зображення варіантів використання (функцій системи зі сторони користувача) та як актори (користувачі або інші системи) взаємодіють з ними. Цей тип діаграм використовується на ранніх стадіях розробки, коли необхідно визначити вимоги до системи та її функціональності [14].

Основними компонентами діаграми є актор, який визначається як користувач або інша система, що взаємодіє з системою, і варіант використання, який визначається як дія або набір дій, які система виконує у відповідь на запит актора. Встановлені зв’язки між цими сутностями окреслюють доступні дії для даного актора. Також можуть використовуватися спеціальні зв’язки, такі як включення, розширення та узагальнення, які використовуються для узагальнення акторів або сценаріїв [14].

Діаграми варіантів використання сприяють встановленню спільного розуміння між замовниками, аналітиками та розробниками щодо очікуваної функціональності системи.

Кожен варіант використання описує дії, які актор може здійснити в системі для реалізації певного сценарію. Ці сценарії відображають різні способи взаємодії з функціоналом системи. Варіанти використання пропонують

системний підхід до формулювання та документування вимог до поведінки системи з точки зору її користувачів або зовнішніх систем [10, 14].

На рисунку 2.1 представлена діаграма варіантів використання паралельного вебсерверу для чисельного моделювання.

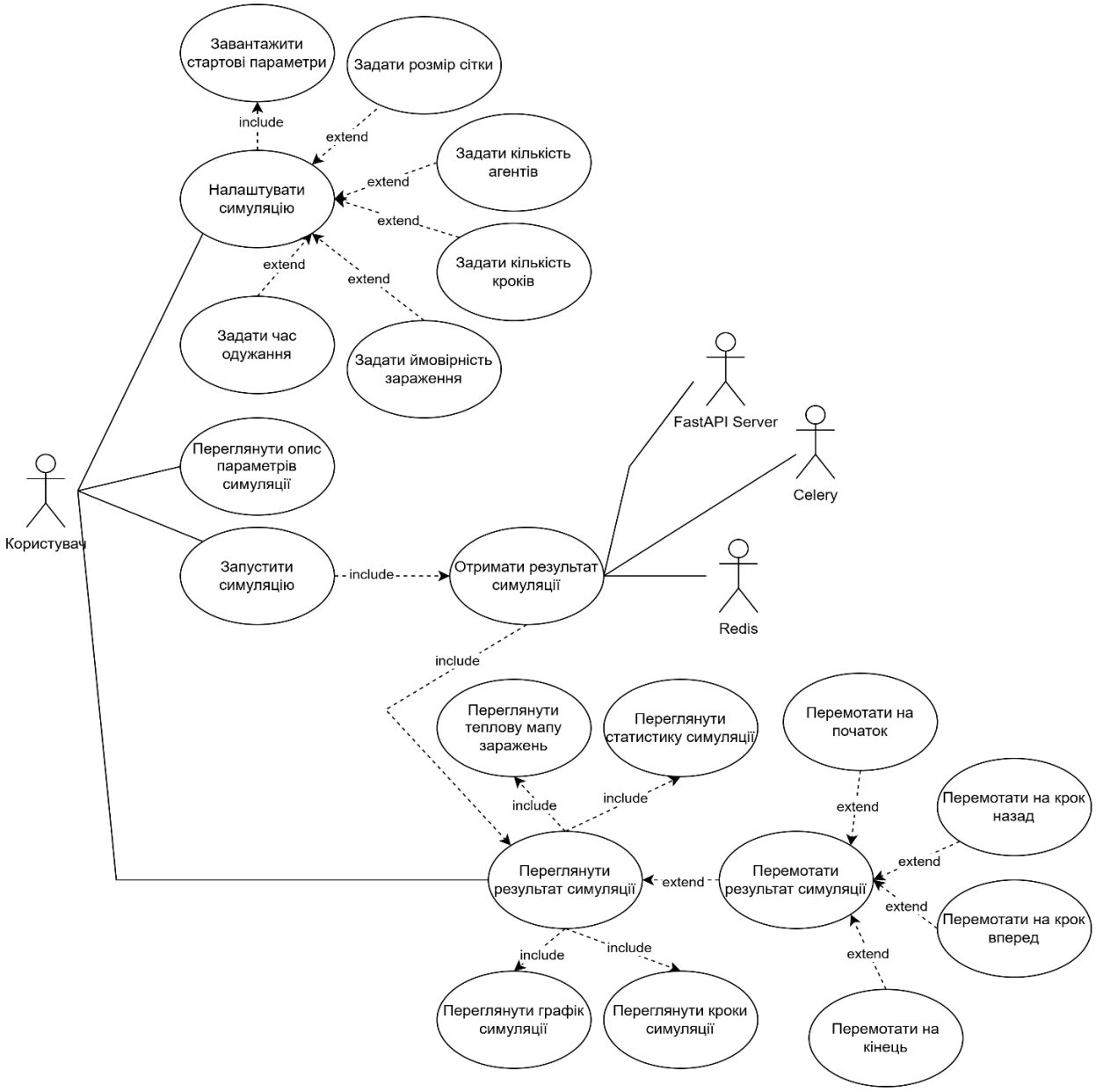


Рисунок 2.1 – Діаграма варіантів використання

На діаграмі представлено акторів «Користувач», який відображає функціональні можливості користувача, а також актори «FastAPI Server», «Redis» та «Celery», які є зовнішніми акторами, що забезпечують отримання результату.

Виділено 1 основний варіант використання – «Запустити симуляцію». Після того, як користувач переходить до інтерфейсу, система відображає форму введення параметрів, які ініціалізує початковими даними. Користувач може змінити парметри симуляції або відразу запустити симуляцію з початковими. Для запуску процесу симуляції, користувач має натиснути на кнопку «Запустити симуляцію». Після надсилання запиту на початок симуляції, система отримує ідентифікатор завдання симуляції і починає кожні 2 секунди опитувати сервер на предмет завершення обчислень. Після успішного обчислення, сервер повертає дані та час симуляції – клієнт відображає результати симуляції у вигляді графіків, таблиць, сіток та числових значень.

2.2.1 Опис варіантів використання

Прецедент «Налаштувати симуляцію».

Призначення: даний варіант використання надає можливість користувачу задати параметри, які необхідні для запуску агентної симуляції.

Основний потік подій: даний варіант використання починає виконуватися після завантаження інтерфейсу. Користувач може змінити початкові значення наступних параметрів: розмір сітки, кількість агентів, кількість кроків, ймовірність зараження та тривалість одужання.

Виняткова ситуація 1: введено нечислове або нульове значення – система підсвічує відповідні поля, відображає помилку валідації та блокує кнопку запуску симуляції.

Виняткова ситуація 2: обов’язкові поля не заповнено – система підсвічує відповідні поля, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Прецедент «Завантажити стартові параметри».

Призначення: даний варіант використання надає можливість користувачу запустити симуляцію без вказання параметрів.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач завантажив інтерфейс. Система автоматично заповнює поля форми значеннями за замовчуванням.

Виняткова ситуація 1: сталася помилка при ініціалізації – система залишає попередні значення і відображає повідомлення про помилку.

Прецедент «Задати розмір сітки».

Призначення: даний варіант використання надає можливість користувачу вказати розмір поля для симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ціле число в полі «Розмір сітки». Система приймає значення і зберігає його в об’єкті форми.

Виняткова ситуація 1: введено нечислове або нульове значення – система показує повідомлення про помилку і блокує кнопку запуску симуляції.

Виняткова ситуація 2: поле не заповнено – система підсвічує поле, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Прецедент «Задати кількість агентів».

Призначення: даний варіант використання надає можливість користувачу задати кількість агентів, які будуть задіяні в симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ціле число в полі «Кількість агентів». Система приймає значення і зберігає його в об’єкті форми.

Виняткова ситуація 1: введено нечислове або нульове значення – система показує повідомлення про помилку і блокує кнопку запуску симуляції.

Виняткова ситуація 2: поле не заповнено – система підсвічує поле, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Прецедент «Задати кількість кроків».

Призначення: даний варіант використання надає можливість користувачу встановити тривалість симуляції в кроках.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ціле число в полі «Кількість кроків». Система приймає значення і зберігає його в об’єкті форми.

Виняткова ситуація 1: введено нечислове або нульове значення – система показує повідомлення про помилку і блокує кнопку запуску симуляції.

Виняткова ситуація 2: поле не заповнено – система підсвічує поле, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Виняткова ситуація 3: значення виходить за межі допустимого діапазону – система показує повідомлення про помилку та блокує кнопку запуску симуляції.

Прецедент «Задати ймовірність зараження».

Призначення: даний варіант використання надає можливість користувачу вказати ймовірність зараження при контакті.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ціле число в полі «Ймовірність зараження». Система приймає значення і зберігає його в об’єкті форми.

Виняткова ситуація 1: введено нечислове або нульове значення – система показує повідомлення про помилку і блокує кнопку запуску симуляції.

Виняткова ситуація 2: поле не заповнено – система підсвічує поле, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Виняткова ситуація 3: значення виходить за межі допустимого діапазону – система показує повідомлення про помилку та блокує кнопку запуску симуляції.

Прецедент «Задати час одужання».

Призначення: даний варіант використання надає можливість користувачу вказати, через скільки кроків агент вважається вилікуваним.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ціле число в полі «Час одужання». Система приймає значення і зберігає його в об’єкті форми.

Виняткова ситуація 1: введено нечислове або нульове значення – система показує повідомлення про помилку і блокує кнопку запуску симуляції.

Виняткова ситуація 2: поле не заповнено – система підсвічує поле, відображає помилку про необхідність заповнення та блокує кнопку запуску симуляції.

Прецедент «Переглянути опис параметрів симуляції».

Призначення: даний варіант використання надає можливість користувачу ознайомитися з описом усіх параметрів симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває розділ «Опис параметрів», система відображає текстове пояснення впливу кожного параметра на процес та результати симуляції.

Виняткова ситуація 1: текстовий блок не завантажився – система приховує розділ.

Прецедент «Запустити симуляцію».

Призначення: даний варіант використання надає можливість користувачу запустити процес чисельної симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на кнопку «Запустити симуляцію». Система перевіряє коректність введених даних, формує запит до сервера і передає параметри для обчислень. Після успішного запуску, система починає опитування на предмет завершення симуляції.

Передумова: усі обов’язкові параметри симуляції мають бути введені.

Виняткова ситуація 1: невалідні параметри – система блокує запуск і виводить повідомлення.

Виняткова ситуація 2: сервер не відповідає або повернув помилку – система повідомляє про неможливість запуску.

Прецедент «Отримати результат симуляції».

Призначення: даний варіант використання реалізує механізм періодичного опитування стану симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли система автоматично відправляє запити до серверу з інтервалом у 2 секунди. Після завершення обчислень, результат отримується і обробляється.

Виняткова ситуація 1: задача не завершена – система повертає відповідний статус та продовжує опитування.

Виняткова ситуація 2: сервер повернув помилку або статус «error» – користувачу виводиться повідомлення про помилку.

Прецедент «Переглянути результат симуляції».

Призначення: даний варіант використання надає можливість користувачу переглянути результат завершеної симуляції в різних формах.

Основний потік подій: даний варіант використання починає виконуватися після завершення симуляції, система обробляє масив даних і відображає: сітку станів за кроками, підсумкову статистику, графік та теплову мапу.

Передумова: симуляція повинна бути успішно завершена.

Виняткова ситуація 1: результат у неправильному форматі – система повідомляє про помилку відображення.

Виняткова ситуація 2: одна або кілька візуалізацій не завантажились – система відображає часткові результати.

Прецедент «Переглянути графік симуляції».

Призначення: даний варіант використання дозволяє користувачу переглянути графік зміни кількості агентів у різних станах.

Основний потік подій: даний варіант використання починає виконуватися, коли система відображає лінійний графік, сформований на основі даних симуляції.

Передумова: результат симуляції має бути обчислений.

Виняткова ситуація 1: дані пошкоджені або не знайдені – система виводить повідомлення про помилку.

Прецедент «Переглянути кроки симуляції».

Призначення: даний варіант використання дозволяє переглядати стан поля на кожному кроці симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач керує перемиканням кроків, система виводить відповідний до кроку стан сітки.

Передумова: результат симуляції має бути обчислений.

Виняткова ситуація 1: вихід за межі кількості кроків – система блокує відповідні кнопки перемикання.

Прецедент «Переглянути теплову мапу заражень».

Призначення: даний варіант використання дозволяє переглянути сумарну інтенсивність зараження по всій сітці.

Основний потік подій: даний варіант використання починає виконуватися, коли система рендерить теплову мапу на основі частоти заражень у кожній клітинці.

Передумова: результат симуляції має бути обчислений.

Виняткова ситуація 1: немає даних – карта не відображається.

Прецедент «Переглянути статистику симуляції».

Призначення: даний варіант використання дає змогу побачити ключові числові метрики (макс. зараження, вилікувані, здорові тощо).

Основний потік подій: даний варіант використання починає виконуватися, коли система показує текстовий блок з числовими результатами.

Передумова: результат симуляції має бути обчислений.

Виняткова ситуація 1: статистика не згенерована – виводиться повідомлення про збій.

Прецедент «Перемотати результат симуляції».

Призначення: даний варіант використання надає можливість користувачу змінювати крок перегляду симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач використовує відповідні кнопки для переходу між кроками симуляції. Система оновлює відображення сітки відповідно до обраного кроку.

Передумова: результат симуляції має бути обчислений.

Виняткова ситуація 1: поточний крок виходить за межі допустимого – система ігнорує дію.

Прецедент «Перемотати на початок».

Призначення: даний варіант використання надає можливість повернутись до першого кроку симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Перемотати на початок» – сітка оновлюється станом на перший крок симуляції.

Передумова: симуляція успішно завершена.

Виняткова ситуація 1: симуляція вже перебуває на першому кроці – дія ігнорується.

Прецедент «Перемотати на кінець».

Призначення: даний варіант використання надає можливість повернутись до фінального кроку симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Перемотати на кінець» – сітка оновлюється станом на фінальний крок симуляції.

Передумова: симуляція успішно завершена.

Виняткова ситуація 1: симуляція вже перебуває на фінальному кроці – дія ігнорується.

Прецедент «Перемотати на крок назад».

Призначення: даний варіант використання надає можливість повернутись до попереднього відносно поточного кроку симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Попередній крок» – сітка оновлюється станом на попередній крок симуляції.

Передумова: симуляція успішно завершена.

Виняткова ситуація 1: симуляція вже перебуває на першому кроці – дія ігнорується.

Прецедент «Перемотати на крок вперед».

Призначення: даний варіант використання надає можливість повернутись до наступного відносно поточного кроку симуляції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Наступний крок» – сітка оновлюється станом на наступний крок симуляції.

Передумова: симуляція успішно завершена.

Виняткова ситуація 1: симуляція вже перебуває на фінальному кроці – дія ігнорується.

2.3 Діаграма діяльності

Діаграма діяльності в UML – це інструмент, який використовується для моделювання бізнес-процесів, алгоритмів і потоків управління в системі. Вона описує послідовність дій, що виконуються під час певного процесу або варіанту використання. Цей тип діаграм виділяється інтуїтивно-зрозумілою структурою, а тому їх часто використовують на етапах аналізу та проєктування системи [14].

Основні компоненти діаграми включають вузли діяльності, потоки управління, вузли рішень, вузли об’єднання та умови. Також можна використовувати потоки обробки даних, початкові та кінцеві точки і паралельні гілки (вузли розгалуження/з’єднання) для моделювання одночасного виконання декількох процесів. Це дозволяє моделювати як прості алгоритми, так і складні сценарії, що включають розгалуження і паралельну обробку [10].

На рисунках 2.2 – 2.3 наведено діаграму діяльності прецеденту «Запустити симуляцію».

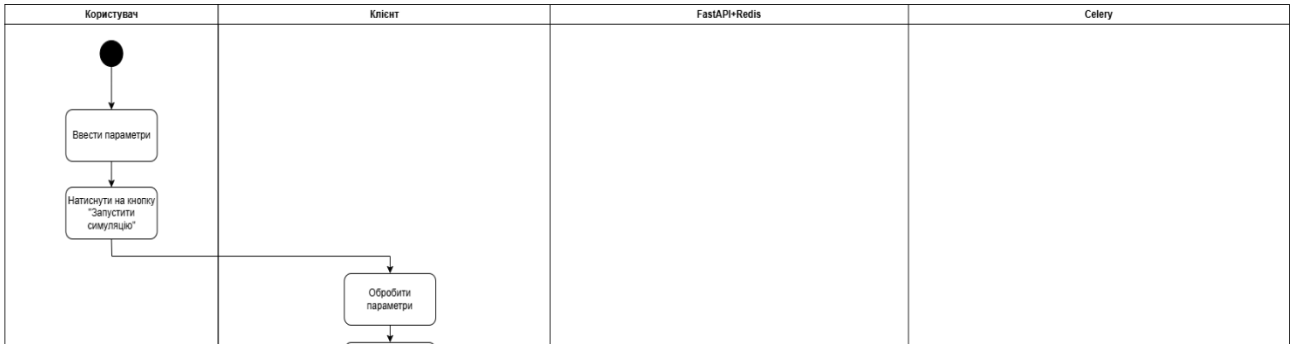


Рисунок 2.2 – Діаграма діяльності (1 частина)

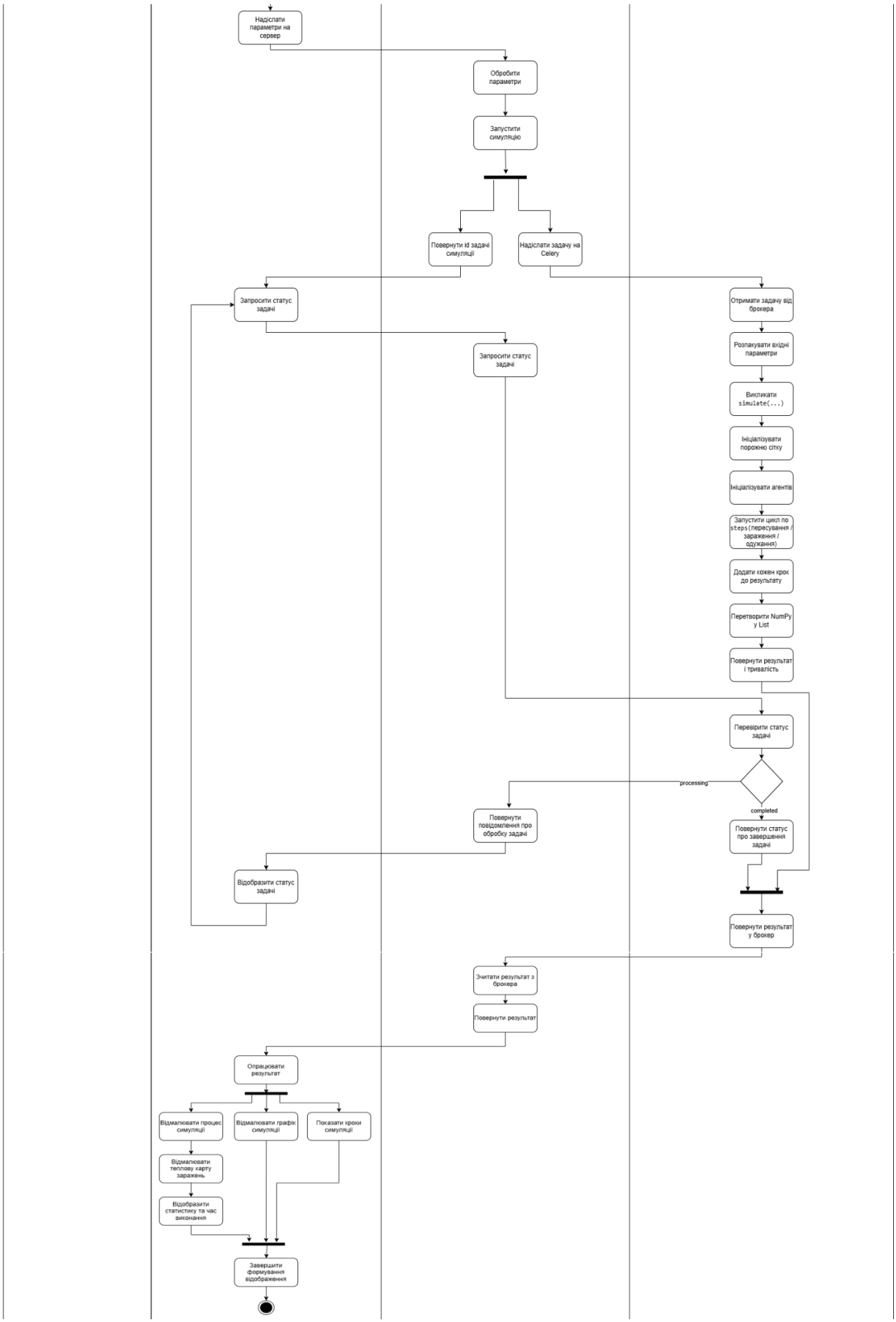


Рисунок 2.3 – Діаграма діяльності (2 частина)

Діаграми діяльності є особливо ефективним інструментом для візуалізації логіки операцій або бізнес-процесів, де розуміння не тільки «що» функціонує в системі, але й «як» і «в якій послідовності» є першорядним.

2.4 Діаграма послідовності

Діаграма послідовності, тип діаграми взаємодії в UML, використовується для моделювання послідовності повідомлень, якими обмінюються об’єкти або компоненти системи під час виконання певного сценарію. Цей тип діаграми дозволяє побачити, хто з ким і в якій послідовності взаємодіє під час виконання певного функціоналу [14].

На рисунку 2.4 описана діаграма послідовності прецеденту «Запустити симуляцію».

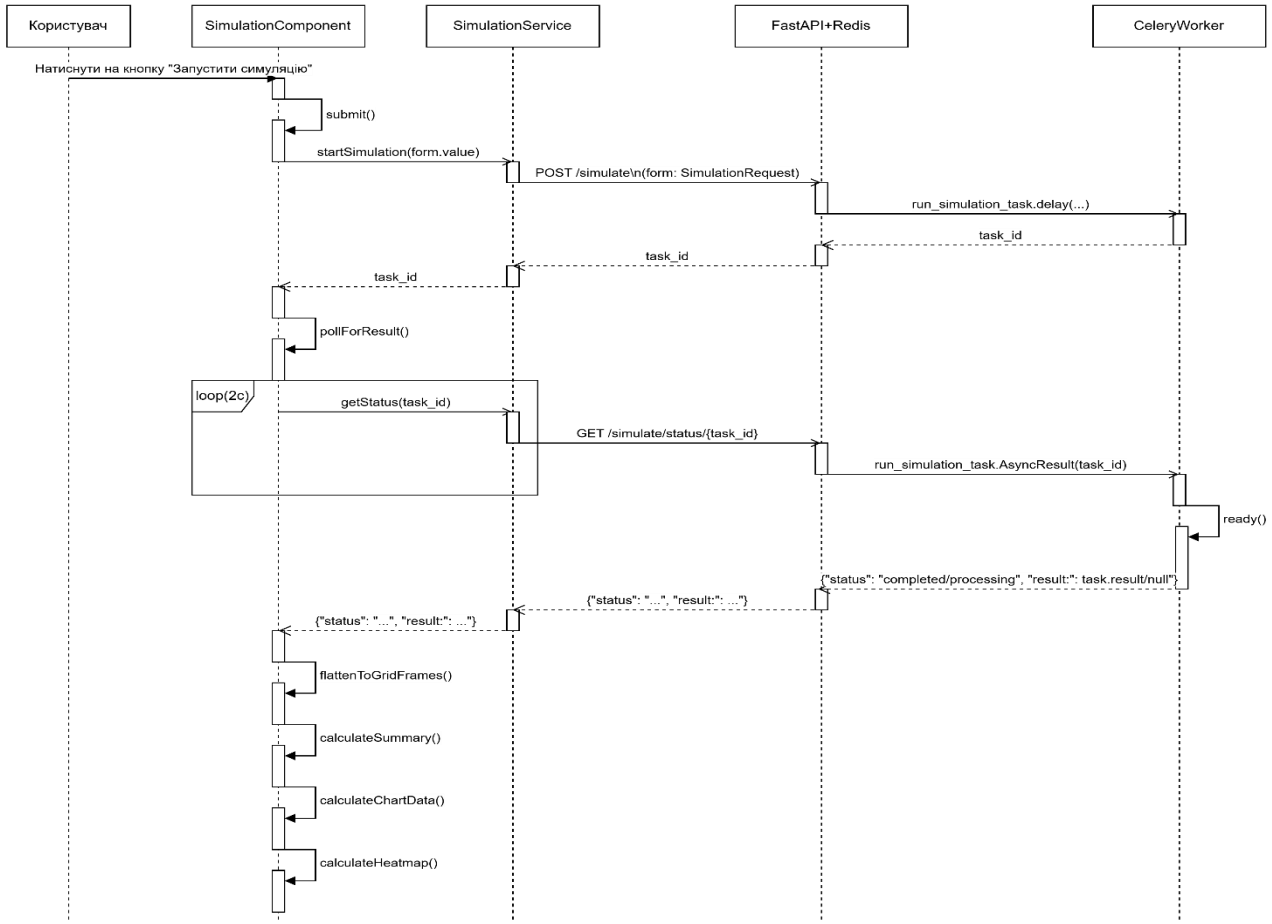


Рисунок 2.4 – Діаграма послідовності

Основними компонентами діаграми є об’єкти, які можуть бути акторами або компонентами системи, розташовані по горизонталі, та лінії життя, які розташовуються по вертикалі і демонструють «життя» об’єкта під час виконання сценарію. Повідомлення, які можуть бути визначені як виклики методів або сигнали, передаються стрілками між об’єктами і відображаються зверху вниз у хронологічному порядку. Умовні блоки (alt, opt, loop) також можуть використовуватися для представлення альтернатив або повторень дій [10].

Діаграми послідовності є ефективним інструментом для ілюстрації взаємодії між компонентами системи під час реалізації варіантів використання, що полегшує розробникам розуміння послідовності викликів, залежностей між модулями та потенційних областей для оптимізації або помилок у логіці. Ці діаграми використовуються також на етапі тестування, оскільки полегшують генерацію тестів, які відображають стандартну поведінку системи.

2.5 Діаграма розгортання

Діаграма розгортання в UML використовується для моделювання фізичної структури системи, включаючи розгортання програмних компонентів (виконуваних модулів, сервісів тощо) на апаратному забезпеченні [10].

Первинними компонентами діаграми є вузли, які визначаються як фізичні пристрої або середовища виконання (наприклад, сервер, клієнт, база даних, віртуальна машина тощо). Другим компонентом є артефакти, які представляють програмне забезпечення або його компоненти, що розгорнуті на цих вузлах. Взаємозв’язки між вузлами, представлені лініями, що вказують на мережеві взаємодії, ілюструють наявність фізичних або логічних каналів зв’язку [10].

Діаграма розгортання має особливе значення на етапі проєктування архітектури системи та при плануванні впровадження системи. Цей підхід дозволяє візуалізувати роботу системи в реальному середовищі, що полегшує врахування таких важливих факторів, як інфраструктура, продуктивність, безпека та вимоги до масштабованості [14].

На рисунку 2.5 наведено діаграму розгортання системи.

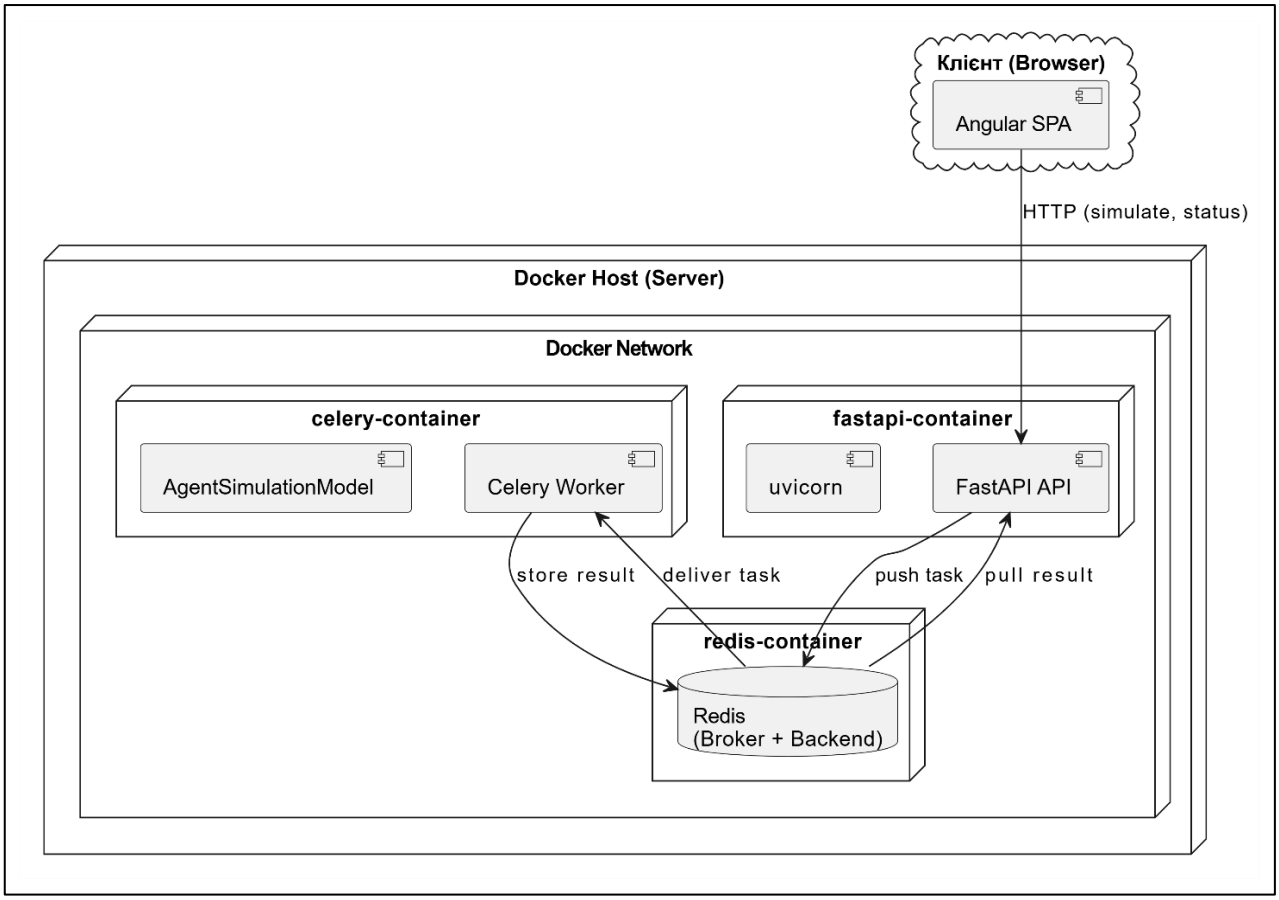


Рисунок 2.5 – Діаграма розгортання

Підсумовуючи, у даному розділі було закладено фундаментальні принципи майбутньої системи. Представлені діаграми сформували чітке бачення функціональних можливостей та деталізували внутрішню логіку та потоки даних. Завдяки такому комплексному підходу маємо повне уявлення про архітектуру та поведінку системи. Цей етап є критично важливим містком між аналізом вимог та безпосередньою реалізацією, забезпечуючи, що кожен аспект системи буде продуманий та оптимізований перед початком кодування.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Для реалізації системи було використано стек технологій, що дозволяє організувати клієнт-серверну архітектуру з підтримкою паралельних обчислень.

Angular – SPA-фреймворк для створення вебзастосунків, який виділяється реактивним підходом до побудови інтерфейсів, модульністю та інкапсуляції логіки компонента [1].

RxJS – бібліотека для реактивного програмування, яка використовується в Angular. Дозволяє реалізувати асинхронний обмін даними за допомогою об’єктів типу Observable [6, 20].

FastAPI – високопродуктивний фреймворк для створення REST API на Python. Підтримує асинхронність, валідацію вхідних даних через Pydantic та має зручну інтеграцію з Celery [9].

Celery – асинхронна система керування задачами, що дозволяє відокремити процес обчислень від основного API-сервера. Працює у парі з брокером Redis [4].

Redis – система зберігання типу «ключ-значення», яка виступає як брокер повідомлень і сховище результатів для Celery [19].

Docker – контейнеризація серверної частини проєкту, що забезпечує зручне розгортання та масштабування [8].

3.2 Основні класи системи

Основним прецедентом системи є «Запустити симуляцію», тому базові класи системи забезпечують налаштування параметрів симуляції, обчислення на сервері, отримання результату та візуалізацію даних зі сторони клієнту.

Головними класами системи зі сторони клієнту є:

– SimulationComponent – головний Angular-компонент, який відповідає за ініціалізацію симуляції, взаємодію з формою, надсилання запитів до сервера, обробку результатів, та передачу їх у компоненти візуалізації [1, 3];
– SimulationService – сервіс, що відповідає за обмін даними між клієнтом та сервером. Містить методи startSimulation() та getStatus() для роботи з REST API [1, 3];
– GridViewComponent – компонент, що відповідає за покрокове відображення станів симуляції у вигляді сітки [1, 3];
– HeatmapViewComponent – компонент, який будує теплову мапу заражень на основі кумулятивної частоти появи інфікованих агентів [1, 3].

Головними класами та функціями зі сторони сервера є:

– SimulationRequest (Pydantic) – модель, яка визначає схему вхідних параметрів симуляції з боку API [9, 15];
– run_simulation_task (Celery task) – задача, що приймає параметри, викликає симуляцію та повертає результат [4, 9, 15];
– AgentSimulationModel – реалізує безпосередньо логіку симуляції, збереження станів, зараження та відновлення агентів по кроках [4, 9].

Нижче представлено приклад функції симуляції на стороні бекенду, яка використовується всередині задачі Celery (рис. 3.1).

```
def simulate(grid_size, agent_count, steps, infection_prob, recovery_steps):
    grid = initialize_grid(grid_size, agent_count)
    frames = []
    for _ in range(steps):
        update_infections(grid, infection_prob, recovery_steps)
        frames.append(copy_grid(grid))
    return frames
```

Рисунок 3.1 – Функція симуляції на сервері

Детальніше ознайомитися з кодом SimulationComponent можна в додатку А, з SimulationService – у додатку Б, а код AgentSimulationModel наведено в додатку В.

3.3 Встановлення та налаштування оточення

Перед початком роботи необхідно встановити та налаштувати оточенння. Так як використовується клієнт-серверна архітектура, то необхідно розгорнути оточення, як зі сторони серверу, так і зі сторони клієнту.

Почнемо з серверної частини, для початку необхідно встановити Python3 та Redis Server, робимо це за допомогою команди sudo apt install redis-server python3 (рис. 3.2) [15, 19, 21].

```
svarog@svarog:~$ sudo apt install redis-server python3
[sudo] пароль до svarog:
Зчитування переліків пакунків... Виконано
Побудова дерева залежностей... Виконано
Зчитування інформації про стан... Виконано
redis-server вже є новою версією (5:7.0.15-1ubuntu0.24.04.1).
python3 вже є новою версією (3.12.3-0ubuntu2).
Наступні пакунки були встановлені автоматично і більше не потрібні:
  libllvm17t64 libxcb-cursor0 libxcb-xinerama0 libxcb-xtest0 python3-netifaces
Використовуйте 'sudo apt autoremove' щоб видалити їх.
```

Рисунок 3.2 – Встановлення Python3 та Redis Server

Після встановлення створюємо файл із залежностями, який буде використовуватися у Dockerfile (рис. 3.3) [8], та структуру проєкту (рис. 3.4) [9].

requirements.txt

```
1  fastapi
2  uvicorn[standard]
3  celery
4  redis
5  pydantic
6  numpy
7  numba
```

Рисунок 3.3 – Файл requirements.txt

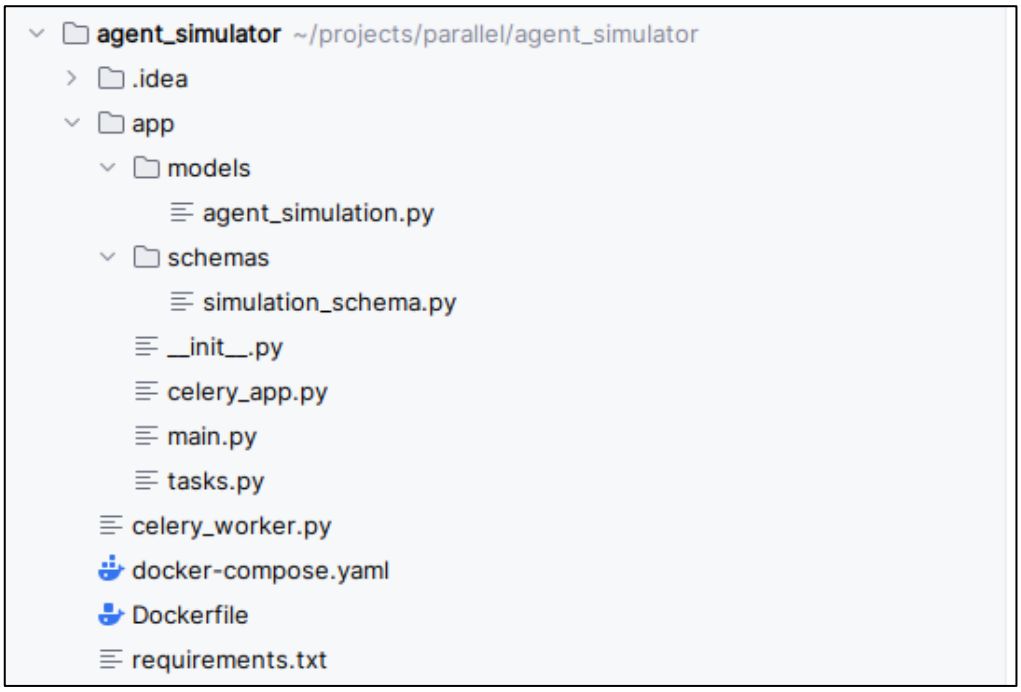


Рисунок 3.4 – Структура серверної частини

Для зборки та запуску fastapi-серверу використовується Dockerfile, який встановлює залежності, надані у файлі requirements.txt, і запускає сервер на 8000 порту (рис. 3.5).

Dockerfile

```
1  FROM python:3.11-slim
2
3  RUN apt-get update && apt-get install -y build-essential && apt-get clean
4  WORKDIR /app
5  COPY . .
6  RUN pip install --upgrade pip
7  RUN pip install -r requirements.txt
8  CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
9
```

Рисунок 3.5 – Dockerfile

Серверна частина проєкту запускається за допомогою docker-compose.yaml. У цьому файлі прописується кількість ядер при запуску Celery, для цього застосовується змінна оточення CELERYD_CONCURRENCY (рис. 3.6) [8]. З повним кодом файлу можна ознайомитися в додатку Г.

```yaml
services:

  celery:
    build: .
    container_name: celery-worker
    command: celery -A celery_worker.celery_app worker --loglevel=info -Q simulation
    depends_on:
      - redis
    volumes:
      - .:/app
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - CELERY_RESULT_BACKEND=redis://redis:6379/0
      - CELERYD_CONCURRENCY=4
```

Рисунок 3.6 – Налаштування Celery у docker-compose.yaml

Запуск контейнерів відбувається за допомогою команди docker compose up –build (рис. 3.7 та рис. 3.8) [8].

```
svarog@svarog:~/projects/parallel/agent_simulator$ docker compose up --build
WARN[0000] /home/svarog/projects/parallel/agent_simulator/docker-compose.yaml: the attribute `version` is obsolete, it will be ignored
[+] Building 35.0s (17/20)
 => [celery internal] load build definition from Dockerfile
 => => transferring dockerfile: 297B
 => [backend internal] load build definition from Dockerfile
 => => transferring dockerfile: 297B
 => [backend internal] load metadata for docker.io/library/python:3.11-slim
 => [backend internal] load .dockerignore
 => => transferring context: 2B
 => [celery internal] load .dockerignore
 => => transferring context: 2B
 => [backend 1/6] FROM docker.io/library/python:3.11-slim@sha256:dbf1de478a55d6763afaa39c2f3d7b54b25230614980276de5cacdde79529d0c
 => => resolve docker.io/library/python:3.11-slim@sha256:dbf1de478a55d6763afaa39c2f3d7b54b25230614980276de5cacdde79529d0c
```

Рисунок 3.7 – Запуск контейнерів

```
celery-worker   | [tasks]
celery-worker   |   . app.tasks.run_simulation_task
celery-worker   |
fastapi-server  | INFO:     Started server process [1]
fastapi-server  | INFO:     Waiting for application startup.
fastapi-server  | INFO:     Application startup complete.
fastapi-server  | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
celery-worker   | [2025-05-23 10:59:12,908: INFO/MainProcess] Connected to redis://redis:6379/0
celery-worker   | [2025-05-23 10:59:12,910: INFO/MainProcess] mingle: searching for neighbors
celery-worker   | [2025-05-23 10:59:13,916: INFO/MainProcess] mingle: all alone
celery-worker   | [2025-05-23 10:59:13,928: INFO/MainProcess] celery@6c81d70def30 ready.


w Enable Watch
```

Рисунок 3.8 – Результати запуску

Так як для клієнтської сторони використовується Angular, то спочатку необхідно встановити npm, а за допомогою нього – Angular CLI. Встановлюється за допомогою команди npm install -g @angular/cli. Встановлення цього пакета надає можливість створити Angular-проєкт за допомогою консольної команди ng new project_name (рис. 3.9) [1].



Рисунок 3.9 – Створення Angular-проєкту

3.4 Опис API

У системі представлено два API-endpoints:

– POST /simulate надсилає запит для запуску симуляції та повертає або ID запущеного завдання, або помилку;
– GET /simulate/status/{task_id} надсилає запит на отримання статусу завдання за його ID. Повертає тільки статус «processing» у випадку, якщо завдання ще оброблюється, статус «completed», дані та час – завдання успішно завершено, або помилку у інших випадках.

Для опису взаємодії з API використано Swagger Hub (рис. 3.10).

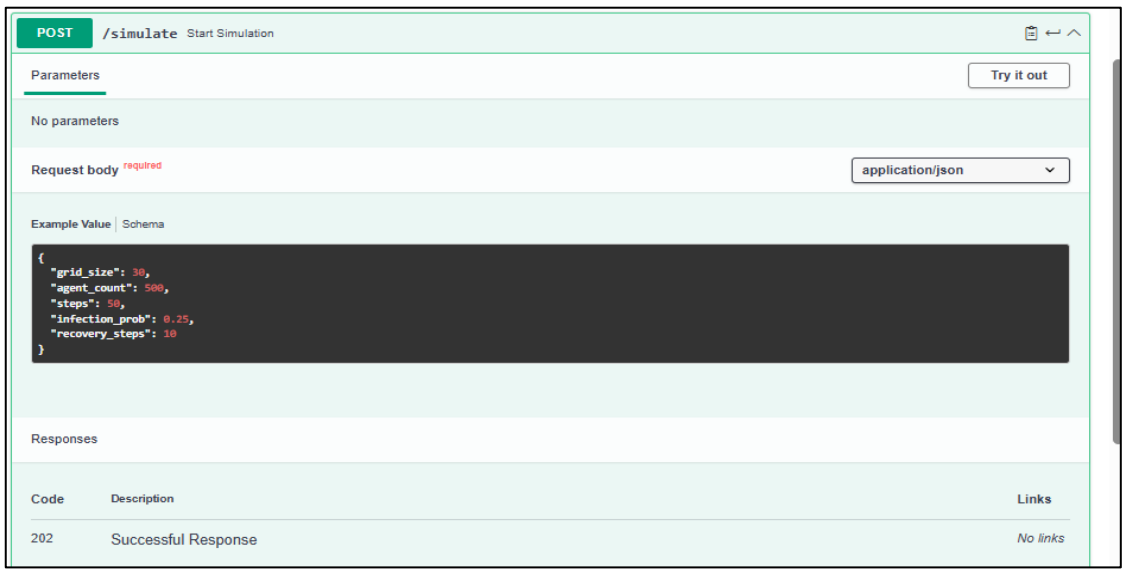


Рисунок 3.10 – Приклад опису API у Swagger

Детально можна ознайомитися за посиланням https://app.swaggerhub.com/apis/zaporizhzhianational/agent-simulation_api/1.0.0.

3.5 Тестування проєкту

Unit-тестування використовується для перевірки роботи окремих одиниць коду, таких як сервіси, пайпи та компоненти, ізольовано від інших частин системи. Для написання unit-тестів використовуються фреймворки Jasmine та Karma, які надаються Angular CLI за замовчуванням [1, 3].

У наступному прикладі надано unit-тест, який перевіряє POST-запит у методі startSimulation() сервісу SimulationService (рис. 3.11).

```
it('should call POST /simulate with correct payload', () :void => {
  const payload = {
    grid_size: 20,
    agent_count: 300,
    steps: 50,
    infection_prob: 0.25,
    recovery_steps: 10
  };

  service.startSimulation(payload).subscribe();

  const req :TestRequest = httpMock.expectOne('/simulate');
  expect(req.request.method).toBe('POST');
  expect(req.request.body).toEqual(payload);
});
```

Рисунок 3.11 – Unit-тест сервісу SimulationService

Інтеграційне тестування дозволяє перевірити роботу кількох компонентів і сервісів разом. Для інтеграційного тестування використовується фреймворк Cypress, який забезпечує end-to-end тестування Angular-додатків [1, 3].

На рисунку 3.12 наведено приклад тесту для SimulationComponent, який перевіряє повний сценарій: заповнення форми, запуск симуляції та появу графіку.

```
describe('Simulation Flow', () :void => {
  it('runs simulation and displays results', () :void => {
    cy.visit('/');

    cy.get('input[formcontrolname="grid_size"]').clear().type('20');
    cy.get('input[formcontrolname="agent_count"]').clear().type('300');
    cy.get('input[formcontrolname="steps"]').clear().type('50');
    cy.get('input[formcontrolname="infection_prob"]').clear().type('0.3');
    cy.get('input[formcontrolname="recovery_steps"]').clear().type('10');

    cy.contains('Start Simulation').click();

    cy.contains('Крок:').should('be.visible');
    cy.get('canvas').should('be.visible');
  });
});
```

Рисунок 3.12 – Інтеграційний тест

Значення як модульного, так і інтеграційного тестування для забезпечення високої якості Angular-додатків є важливим аспектом циклу розробки. Unit-тести забезпечують функціональність окремих модулів при їх ізольованій роботі, в той час як інтеграційні тести перевіряють спільну роботу компонентів програми [1].

Детально ознайомитися з тестами можна в додатку Д.

3.6 Керівництво користувача

3.6.1 Рівень підготовки користувача

Для ефективної роботи з вебдодатком користувач повинен володіти базовими навичками роботи з комп’ютером та браузером. Ознайомлення з керівництвом користувача обов’язкове для розуміння функціональних можливостей системи, її призначення та послідовності роботи з нею.

3.6.2 Відкриття застосунку та ознайомлення з параметрами

Для початку роботи, користувачу необхідно перейти за посиланням або відкрити локальний вебзастосунок у браузері.

При завантаженні сторінки автоматично відображається форма налаштування симуляції (рис. 3.13).

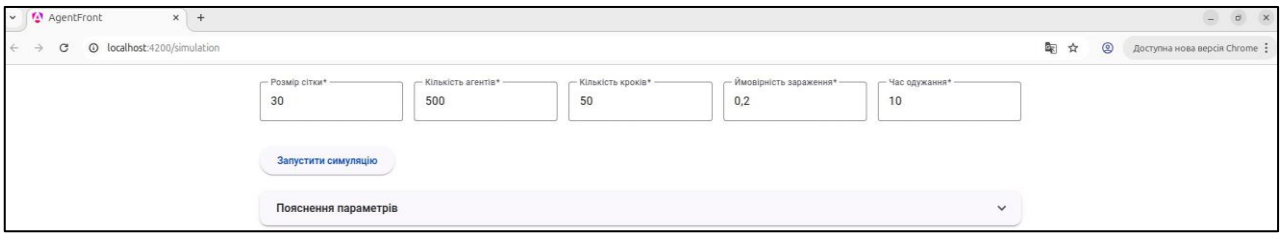


Рисунок 3.13 – Форма налаштування симуляції

Перед запуском симуляції користувач має змогу налаштувати її параметри. Для цього необхідно ввести:

– розмір сітки – кількість клітинок по горизонталі і вертикалі;
– кількість агентів – кількість об’єктів, які будуть брати участь у симуляції;
– кількість кроків – скільки ітерацій (часових кроків) виконуватиметься;
– ймовірність зараження – імовірність, з якою агент інфікує інших;
– час одужання – через скільки кроків агент переходить у стан «вилікуваний».

Користувач може ознайомитися з описом параметрів, перед початком запуску симуляції. Для цього необхідно натиснути на пункт «Пояснення параметрів», який знаходиться під формою налаштування симуляції (рис. 3.14).

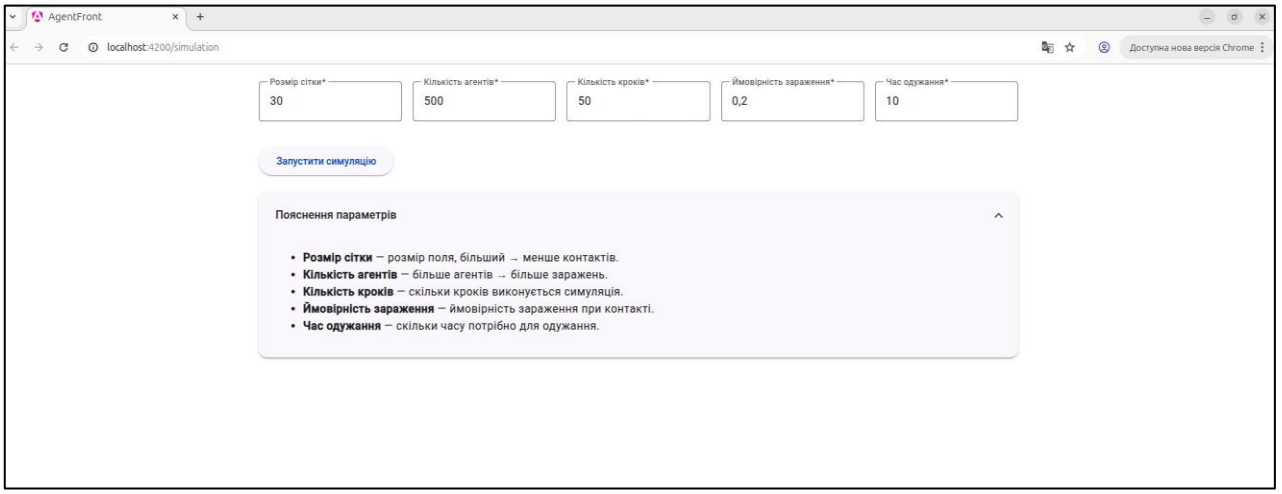


Рисунок 3.14 – Опис параметрів

3.6.3 Запуск симуляції

Після налаштування параметрів користувач натискає кнопку «Запустити симуляцію».

Система надсилає запит до сервера і починає виконання симуляції. На екрані з’являється індикатор завантаження (рис. 3.15).

Після завершення обчислень система автоматично відображає результати.

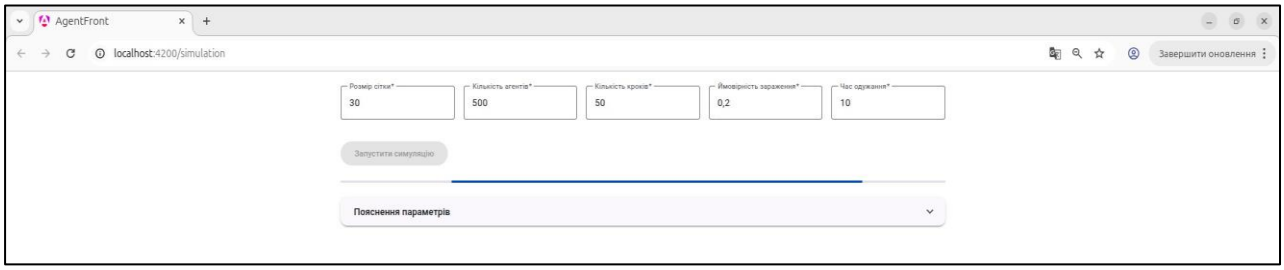


Рисунок 3.15 – Індикатор виконання симуляції

3.6.4 Перегляд результатів симуляції

Після завершення симуляції користувачу стають доступними наступні візуальні блоки:

– сітка станів (Grid) – візуальне представлення кожного кроку симуляції (рис. 3.16);
– графік станів – лінійний графік зміни кількості здорових, інфікованих та вилікуваних агентів (рис. 3.17);
– теплова мапа зараження – показує, які клітинки були інфіковані найчастіше протягом усієї симуляції (рис. 3.18);
– статистика симуляції – підсумкова інформація про пікову кількість заражених, кількість вилікуваних тощо (рис. 3.19).

Також користувач може переглянути таблицю кроків для повного розуміння перебігу симуляції, а також числової візуалізації цього процесу (рис. 3.20).

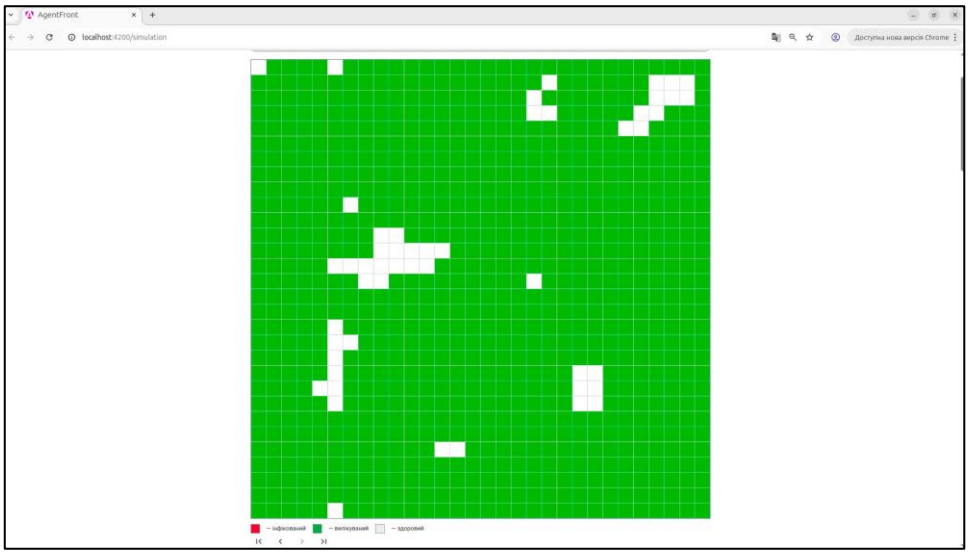


Рисунок 3.16 – Сітка станів на кроці симуляції

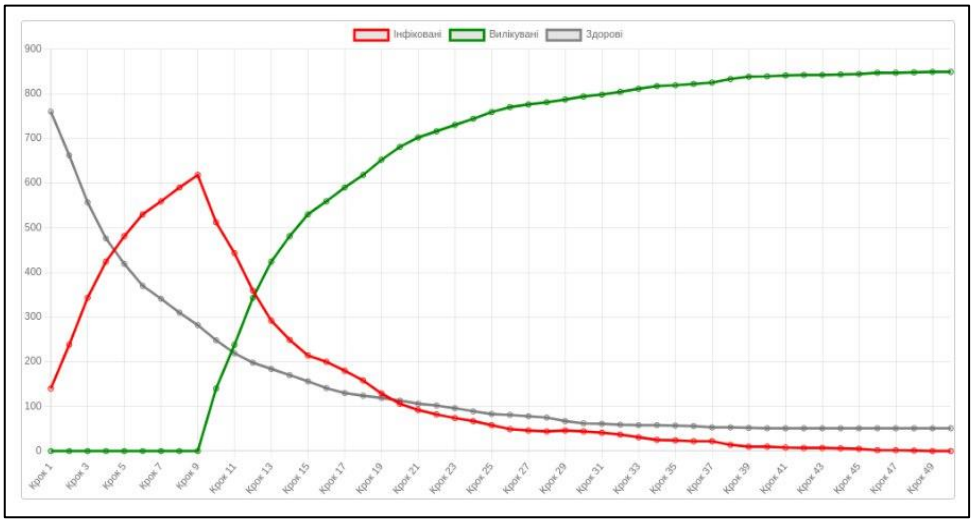


Рисунок 3.17 – Графік динаміки зараження

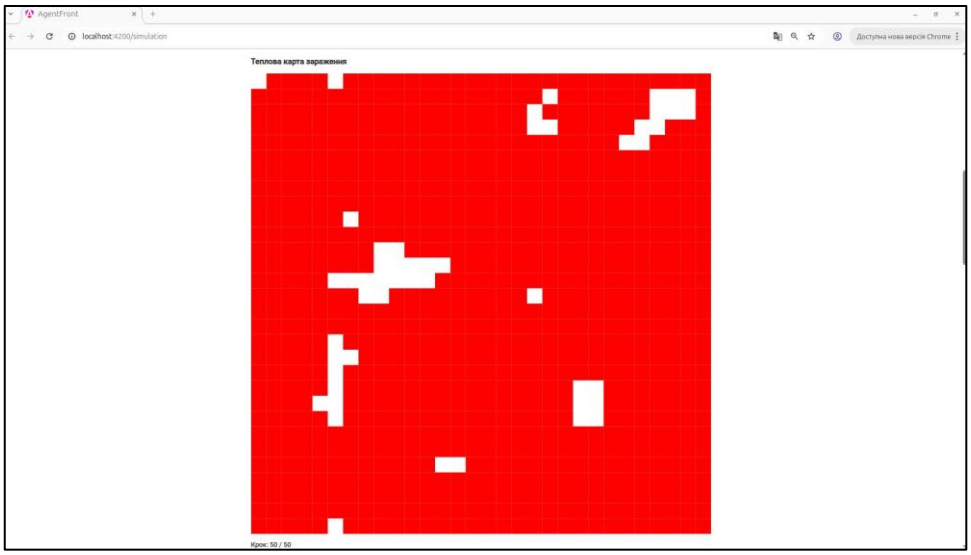


Рисунок 3.18 – Теплова карта зараження



Рисунок 3.19 – Підсумкова статистика симуляції

Крок	Здорові	Інфіковані	Вилікувані
1	760	140	0
2	662	238	0
3	557	343	0
4	476	424	0
5	419	481	0
6	370	530	0
7	341	559	0
8	310	590	0
9	282	618	0
10	248	512	140
11	219	443	238
12	198	359	343
13	184	292	424
14	170	249	481
15	156	214	530
16	141	200	559
17	130	180	590
18	124	158	618
19	119	129	652
20	113	106	681

Рисунок 3.20 – Кроки симуляції

3.6.5 Перегляд по кроках

Користувач може самостійно переміщуватися по кроках симуляції за допомогою кнопок з піктограмами: «На початок», «Назад», «Вперед», «На кінець» (рис. 3.21 – 3.23).

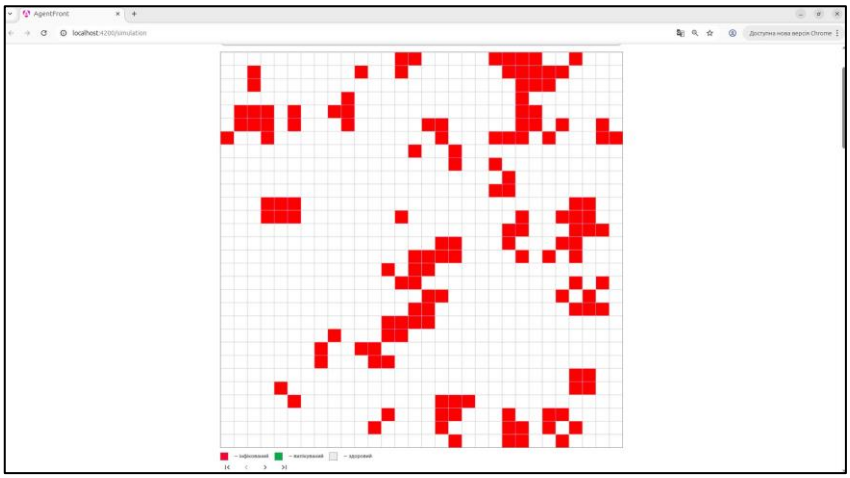


Рисунок 3.21 – Керування перемоткою симуляції (1 крок)

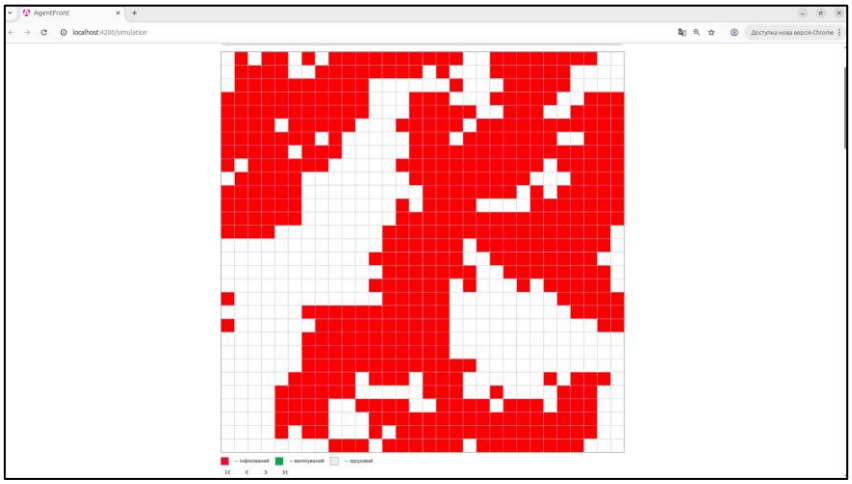


Рисунок 3.22 – Керування перемоткою симуляції (5 крок)

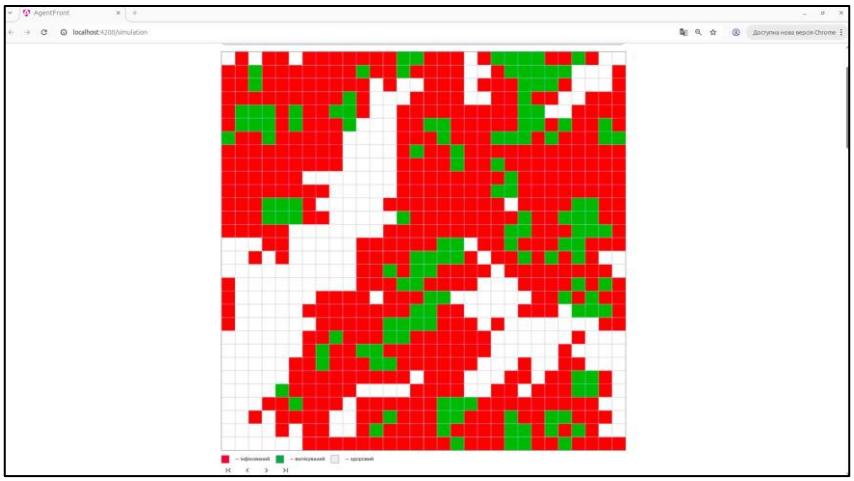


Рисунок 3.23 – Керування перемоткою симуляції (10 крок – початок одужання)

Це дозволяє детально аналізувати, як змінювався стан системи на кожному етапі.

3.6.6 Повторний запуск симуляції

Для того щоб запустити нову симуляцію з іншими параметрами, користувач може змінити значення в полях форми та натиснути «Запустити симуляцію» повторно. При цьому попередні результати будуть очищені автоматично.

На рисунках 3.24 – 3.26 продемонстровано запуск симуляції у якій параметри «Кількість кроків» та «Час одужання» мають однакове значення, що демонструє варіацію симуляції, коли кроків недостатньо для повного одужання.



Рисунок 3.24 – Сітка станів (Повторна симуляція)



Рисунок 3.25 – Теплова карта зараження (Повторна симуляція)

Крок	Здорові	Інфіковані	Вилікувані
1	2834	766	0
2	2461	1139	0
3	2238	1362	0
4	2100	1500	0
5	1984	1616	0
6	1903	1697	0
7	1794	1806	0
8	1753	1847	0
9	1727	1873	0
10	1711	1889	0
11	1705	1895	0
12	1701	1899	0
13	1698	1902	0
14	1697	1903	0
15	1697	1903	0
16	1697	1903	0
17	1697	1903	0
18	1697	1903	0
19	1697	1903	0
20	1697	1137	766

Рисунок 3.26 – Кроки (Повторна симуляція)

Можна побачити, що одужання почалося на 20 кроці, але їх не вистачило для повного одужання всіх агентів.

3.7 Порівняння часу симуляції при різній кількості ядер

Однією з ключових переваг розробленої системи є підтримка паралельних обчислень, що реалізується за допомогою бібліотеки Celery у зв’язці з Redis. Це дозволяє масштабувати виконання симуляцій і зменшувати час обробки задач за рахунок розподілення навантаження між кількома процесами.

Оскільки ефективність симуляції напряму залежить від кількості агентів, розміру сітки та кроків моделювання, доцільно дослідити, як саме змінюється

тривалість виконання при різній кількості доступних обчислювальних ядер. Проведення такого експерименту дозволяє оцінити масштабованість системи та підтвердити доцільність використання паралельної архітектури в рамках обраної задачі чисельного моделювання.

У таблиці 3.1 наведено результати замірів часу виконання симуляцій при різній кількості ядер.

Таблиця 3.1 – Порівняння часу симуляції при різній кількості ядер

Кількість ядер	**Кількість агентів**	**Кількість кроків**	**Час виконання (сек)**
1	500	50	8.4
2	500	50	4.3
4	500	50	2.3
8	500	50	1.4

З отриманих результатів видно, що збільшення кількості ядер суттєво зменшує час виконання симуляції. Найбільший приріст продуктивності спостерігається при переході від 1 до 4 ядер. Проте після 8 ядер приріст сповільнюється через обмеження самої задачі, масштабованість алгоритму та витрати на міжпроцесну синхронізацію [12, 15]. Це демонструє доцільність використання паралельних обчислень у задачах агентного моделювання.

ВИСНОВКИ

У ході виконання даної роботи було розроблено паралельний вебсервер для чисельного моделювання на прикладі агентних систем, який реалізує повноцінну клієнт–серверну архітектуру з підтримкою паралельних обчислень. Система дозволяє запускати симуляції з параметрами, заданими користувачем, а також здійснювати візуальний аналіз результатів за допомогою графіків, анімації, статистики та теплової карти. Архітектура побудована з урахуванням вимог масштабованості, продуктивності та користувацької зручності.

У відповідності з метою кваліфікаційної роботи було реалізовано додаток для чисельного моделювання з використанням таких технологій:

– FastAPI для реалізації серверної логіки та REST API;
– Celery + Redis для фонових паралельних обчислень;
– Angular для створення інтерактивного інтерфейсу;
– Docker для контейнеризації сервісів.

У відповідності з поставленими задачами були виконані наступні етапи створення додатку:

– сформовано функціональні та нефункціональні вимоги, проаналізовано предметну область та оглянуто існуючі аналоги симуляційних систем;
– спроєктовано архітектуру системи, побудовано діаграми варіантів використання, діяльності, послідовності, розгортання; надано детальний опис кожного з прецедентів;
– реалізовано додаток для паралельного чисельного моделювання (наведена інструкція по розгортанню та налаштуваню середовища, опис API, надано керівництво користувача, структуру проєкту та порівняння часу виконання залежно від кількості ядер);
– протестована робота системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. URL: https://angular.io/docs/ (дата звернення: 29.01.2025).

2. AnyLogic. URL: https://www.anylogic.com/ (дата звернення: 26.02.2025).

3. Bampakos A., Deeleman P. Learning Angular: A no-nonsense guide to building web applications with Angular 15. Birmingham : Packt Publishing, 2023. 446 p.

4. Celery Documentation. URL: https://docs.celeryq.dev/en/stable/ (дата звернення: 02.03.2025).

5. Chart.js Documentation. URL: https://www.chartjs.org/docs/latest/ (дата звернення: 02.03.2025).

6. Chebbi L. Reactive Patterns with RxJS for Angular: A practical guide to managing your Angular application’s data reactively and efficiently using RxJS 7. Birmingham : Packt Publishing, 2022. 224 p.

7. Dask. URL: https://www.dask.org/ (дата звернення: 26.02.2025).

8. Docker Manuals. URL: https://docs.docker.com/manuals/ (дата звернення: 05.05.2025).

9. FastAPI Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 02.03.2025).

10. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2003. 208 p.

11. Klügl F., Bazzan A. L. C. Agent-Based Modeling and Simulation. *AI Magazine*. 2012. Vol. 33, Iss. 3. P. 29–40. URL: https://doi.org/10.1609/aimag.v33i3.2425 (дата звернення: 02.03.2025).

12. Soliman H. M. Parallel and Distributed Simulation: Methodologies and Techniques. *Journal of King Saud University – Computer and Information Sciences*. 1998. Vol. 10. P. 27–51. URL: https://doi.org/10.1016/S1319-1578(98)80003-7 (дата звернення: 02.03.2025).

13. Mesa: Agent-based modeling in Python. URL: https://mesa.readthedocs.io/stable/ (дата звернення: 25.02.2025).

14. Mrzyglocki F. UML Summarized: Key Concepts and Diagrams for Software Engineers, Architects, and Designers. Phillip Mrzyglocki, 2023. 59 p.

15. Nelli F. Parallel and High Performance Programming with Python: Unlock parallel and concurrent programming in Python using multithreading, CUDA, Pytorch and Dask. Delhi : AVA, 2023. 392 p.

16. NetLogo. URL: https://ccl.northwestern.edu/netlogo/ (дата звернення: 25.02.2025).

17. ng2-charts Documentation. URL: https://valor-software.com/ng2-charts/ (дата звернення: 30.04.2025).

18. Ray Documentation. URL: https://docs.ray.io/en/latest/ (дата звернення: 26.02.2025).

19. Redis Documentation. URL: https://redis.io/docs/latest/ (дата звернення: 02.03.2025).

20. RxJS Documentation. URL: https://rxjs.dev/api (дата звернення: 30.04.2025).

21. Severance C. R. Python for Everybody: Exploring Data in Python 3. CreateSpace Independent Publishing Platform, 2016. 245 p.

ДОДАТОК А

SimulationComponent

```
import { Component } from '@angular/core';
import {FormBuilder, FormGroup, FormsModule, ReactiveFormsModule, Validators} from
'@angular/forms';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatButtonModule} from '@angular/material/button';
import {MatInputModule} from '@angular/material/input';
import {CommonModule} from '@angular/common';
import {SimulationService} from '../services/simulation.service';
import {GridViewComponent} from '../grid-view/grid-view.component';
import {Summary} from "../models/summary.model";
import {MatSliderModule} from "@angular/material/slider";
import {MatExpansionModule} from "@angular/material/expansion";
import {BaseChartDirective} from "ng2-charts";
import {MatTableModule} from "@angular/material/table";
import {HeatmapViewComponent} from "../heatmap-view/heatmap-view.component";
import {MatIconModule} from '@angular/material/icon';
import {MatProgressBarModule} from '@angular/material/progress-bar';

@Component({
  selector: 'app-simulation',
  standalone: true,
  imports: [
    CommonModule, FormsModule,
    MatFormFieldModule, MatButtonModule, MatInputModule,
    GridViewComponent, MatSliderModule, MatExpansionModule,
    BaseChartDirective, MatTableModule, ReactiveFormsModule,
    HeatmapViewComponent, MatIconModule, MatProgressBarModule
  ],
  templateUrl: './simulation.component.html',
  styleUrl: './simulation.component.scss'
})
export class SimulationComponent {
  form: FormGroup;
```

```
taskId: string | null = null;
frames: number[][][] = [];
currentFrameIndex = 0;
loading = false;
interval: any;
paused = false;
duration: number | null = null;
summary: Summary | null = null;
heatmap: number[][] = [];

chartData: {
  healthy: number[];
  infected: number[];
  recovered: number[];
} = {
  healthy: [],
  infected: [],
  recovered: []
};

constructor(private fb: FormBuilder, private simService: SimulationService) {
  this.form = this.fb.group({
    grid_size: [30, Validators.required],
    agent_count: [500, Validators.required],
    steps: [50, Validators.required],
    infection_prob: [0.2, Validators.required],
    recovery_steps: [10, Validators.required]
  });
}

submit() {
  if (this.form.invalid) return;

  this.frames = [];
  this.summary = null;
  this.duration = null;
  this.chartData = { healthy: [], infected: [], recovered: [] };
  this.loading = true;
```

```
  this.simService.startSimulation(this.form.value).subscribe(res => {
    this.taskId = res.task_id;
    this.pollForResult();
  });
}

pollForResult() {
  this.interval = setInterval(() => {
    if (!this.taskId) return;
    this.simService.getStatus(this.taskId).subscribe(res => {
      if (res.status === 'completed' && Array.isArray(res.result)) {
        clearInterval(this.interval);
        this.frames = this.flattenToGridFrames(res.result);
        this.duration = res.duration;
        this.summary = this.computeSummary(this.frames);
        this.calculateChartData(this.frames);
        this.calculateHeatmap()
        this.loading = false;
        this.playAnimation();
      }
    });
  }, 2000);
}

flattenToGridFrames(flattened: number[][]): number[][][] {
  const size = this.form.value.grid_size;
  return flattened.map(flat => {
    const grid: number[][] = [];
    for (let i = 0; i < size; i++) {
      grid.push(flat.slice(i * size, (i + 1) * size));
    }
    return grid;
  });
}

playAnimation() {
  this.currentFrameIndex = 0;
  this.paused = false;
```

```
    clearInterval(this.interval);
    this.interval = setInterval(() => {
      if (!this.paused && this.currentFrameIndex < this.frames.length - 1) {
        this.currentFrameIndex++;
      } else if (!this.paused) {
        clearInterval(this.interval);
      }
    }, 100);

  }

  calculateHeatmap(): void {
    const size = this.form.value.grid_size;
    const heat = Array.from({ length: size }, () => Array(size).fill(0));

    this.frames.forEach(grid => {
      for (let y = 0; y < size; y++) {
for (let x = 0; x < size; x++) {
          if (grid[y][x] === 1) heat[y][x]++;
        }
      }
    });

    this.heatmap = heat;
  }

  computeSummary(frames: number[][][]): Summary {
    let peakInfected = 0;
    let peakStep = 0;
    let finalRecovered = 0;
    let finalHealthy = 0;

    frames.forEach((grid, index) => {
      let infected = 0;
      grid.forEach(row => row.forEach(val => {
        if (val === 1) infected++;
      }));
      if (infected > peakInfected) {
```

```
      peakInfected = infected;
      peakStep = index;
    }
  });

  const lastGrid = frames[frames.length - 1];
  lastGrid.forEach(row => row.forEach(val => {
    if (val === 2) finalRecovered++;
    if (val === 0) finalHealthy++;
  }));

  return { peakInfected, peakStep, finalRecovered, finalHealthy };
}

calculateChartData(frames: number[][][]) {
  this.chartData = { healthy: [], infected: [], recovered: [] };
  frames.forEach(grid => {
    let h = 0, i = 0, r = 0;
    grid.forEach(row => row.forEach(val => {
      if (val === 0) h++;
      if (val === 1) i++;
      if (val === 2) r++;
    }));
    this.chartData.healthy.push(h);
    this.chartData.infected.push(i);
    this.chartData.recovered.push(r);
  });
}

get frameLabels(): string[] {
  return this.chartData.infected.map((_, i) => `Крок ${i + 1}`);
}

get tableData(): { step: number; healthy: number; infected: number; recovered: number }[] {
  return this.chartData.infected.map((_, i) => ({
    step: i + 1,
    healthy: this.chartData.healthy[i],
    infected: this.chartData.infected[i],
```

```
      recovered: this.chartData.recovered[i]
    }));
  }

  prevStep() {
    if (this.currentFrameIndex > 0) this.currentFrameIndex--;
  }

  nextStep() {
    if (this.currentFrameIndex < this.frames.length - 1) this.currentFrameIndex++;
  }

}
```

ДОДАТОК Б

SimulationService

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { SimulationRequest } from '../models/simulation-request.model';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class SimulationService {
  private apiUrl = 'http://localhost:8000';

  constructor(private http: HttpClient) {}

  startSimulation(data: SimulationRequest): Observable<{ task_id: string }> {
    return this.http.post<{ task_id: string }>(`${this.apiUrl}/simulate`, data);
  }

  getStatus(taskId: string): Observable<{ status: string, result: number[][], duration: number }> {
    return this.http.get<{ status: string, result: number[][], duration: number }>(
      `${this.apiUrl}/simulate/status/${taskId}`
    );
  }
}
```

ДОДАТОК В

AgentSimulationModel

```
import numpy as np
from numba import njit

@njit
def simulate(grid_size: int, agent_count: int, steps: int, infection_prob: float, recovery_steps: int):
    grid = np.zeros((grid_size, grid_size), dtype=np.int8)
    agents = np.random.randint(0, grid_size, size=(agent_count, 2))
    infected = np.random.choice(agent_count, size=agent_count // 10, replace=False)
    infection_time = np.full((grid_size, grid_size), -1, dtype=np.int16)

    for i in infected:
        x, y = agents[i]
        grid[x, y] = 1
        infection_time[x, y] = 0

    results = []

    for step in range(steps):
        for i in range(agent_count):
            x, y = agents[i]
            if grid[x, y] == 1:
                for dx in [-1, 0, 1]:
                    for dy in [-1, 0, 1]:
                        nx, ny = (x + dx) % grid_size, (y + dy) % grid_size
                        if grid[nx, ny] == 0 and np.random.rand() < infection_prob:
                            grid[nx, ny] = 1
                            infection_time[nx, ny] = 0

        for x in range(grid_size):
            for y in range(grid_size):
                if infection_time[x, y] >= 0:
                    infection_time[x, y] += 1
                    if infection_time[x, y] >= recovery_steps:
```

```
                grid[x, y] = 2

    results.append(np.copy(grid))

return results
```

ДОДАТОК Г

Docker compose

```
version: '3.8'

services:
  backend:
    build: .
    container_name: fastapi-server
    ports:
      - "8000:8000"
    volumes:
      - .:/app
    depends_on:
      - redis
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - CELERY_RESULT_BACKEND=redis://redis:6379/0
  celery:
    build: .
    container_name: celery-worker
    command: celery -A celery_worker.celery_app worker --loglevel=info -Q simulation
    depends_on:
      - redis
    volumes:
      - .:/app
    environment:
      - CELERY_BROKER_URL=redis://redis:6379/0
      - CELERY_RESULT_BACKEND=redis://redis:6379/0
      - CELERYD_CONCURRENCY=4
  redis:
    image: redis:7
    container_name: redis
    ports:
      - "6379:6379"
```

ДОДАТОК Д

Angular-тести

```
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { SimulationService } from './simulation.service';

describe('SimulationService', () => {
  let service: SimulationService;
  let httpMock: HttpTestingController;
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [SimulationService]
    });
    service = TestBed.inject(SimulationService);
    httpMock = TestBed.inject(HttpTestingController);
  });
  it('should call POST /simulate with correct payload', () => {
    const payload = {
      grid_size: 20,
      agent_count: 300,
      steps: 50,
      infection_prob: 0.25,
      recovery_steps: 10
    };
    service.startSimulation(payload).subscribe();
    const req = httpMock.expectOne('/simulate');
    expect(req.request.method).toBe('POST');
    expect(req.request.body).toEqual(payload);
  });
  afterEach(() => {
    httpMock.verify();
  });
});
```