EEMCS, TCS
University of Twente
Enschede, Netherlands

EpiNet AI: An Agentic Web Platform for Network-Based Epidemic Simulation Workflows

Created by Design Group 23:

Daria Hesson
Konstantinos Zygouris
Daniel Murse-Caraian
Gela Tsuladze
Ashkan Nikjouyan

Supervisors/Clients:
Mahboobeh Zangiabady: m.zangiabady@utwente.nl
Alberto García Robledo: agarcia@centrogeo.edu.mx

April 17, 2026

UNIVERSITY OF TWENTE.

Contents

Abstract

This report serves as part of the "Design Project" course of the bachelor's degree in Technical Computer Science at the University of Twente. Network-based epidemic modeling has become an important tool for understanding and forecasting the spread of infectious diseases. This project proposes the design and implementation of EpiNet AI, an agentic AI-driven system and web-based dashboard that connects natural language interaction and network-based epidemic simulation. The system will treat NIPA as a validated black-box scientific component and build an intelligent layer around it. In this way, users can express high-level analytical goals in natural language through the web interface, and the AI agent will translate these goals into structured, reproducible NIPA simulation workflows.

Keywords: network-based epidemic modeling, forecasting the spread of infectious diseases, agentic AI, dashboard.

Chapter 1

Introduction

This project explores the integration of advanced epidemic modeling with modern AI techniques to create a more accessible and interactive analytical system. It focuses on three key components: the Network Inference-based Prediction Algorithm (NIPA) for epidemic forecasting, Large Language Models (LLMs) for interpreting user goals and orchestrating workflows, and the Model Context Protocol (MCP) for safe and reliable communication between the AI agent and computational tools. By combining these technologies, the project aims to make sophisticated epidemic modeling methods easier to use, more transparent, and reproducible for a broader audience.

1.1 Technical background

In this section, we introduce the main technologies used in this project and a technical background to understand their functionality.

1.1.1 What is the Network Inference-based Prediction Algorithm?

The Network Inference-based Prediction Algorithm (NIPA) is a machine learning approach that reconstructs latent infection interaction networks from epidemic time-series data and uses these inferred networks to predict future infection dynamics. By modeling how individuals influence one another during an outbreak, NIPA enables more accurate epidemic forecasting compared to purely statistical approaches. However, NIPA and similar models typically require script-based exper-

imentation, manual parameter tuning, and substantial domain expertise, limiting their accessibility and reproducibility outside specialized research environments.

1.1.2 What are Large Language Models?

Large Language Models (LLMs) are advanced neural networks trained on large collections of text to understand and generate human-like language. Besides text generation, they are capable of structured reasoning and tool use. Through an agentic framework, an LLM can interpret high-level user instructions, decompose them into executable steps, call external computational tools, inspect intermediate results, and iteratively refine its strategy.

1.1.3 What is a Model Context Protocol?

Model Context Protocol (MCP) is a structured communication protocol that allows an LLM to safely and reliably interact with external tools, services, or computational systems. It defines explicit tool interfaces with validated inputs and outputs, ensuring controlled, reproducible execution of tasks. In layered architectures, MCP acts as a secure intermediary between the AI agent and the underlying scientific or software components.

1.2 Motivation

Advanced epidemic modeling methods such as NIPA offer powerful tools for forecasting disease spread, but their technical complexity limits accessibility and practical use. The motivation for this project is to make network-based epidemic analysis more transparent, reproducible, and user-friendly by combining an agentic AI with a web-based interface.

Chapter 2

Problem statement

In this chapter, we discuss the problem addressed in this project, the proposal AI-driven platform for orchestrating NIPA simulations, and the potential risks and challenges related to AI interpretation, system integration, usability, and project management

2.1 Problem description

Although NIPA can accurately deduce infection interaction networks and predict outbreak dynamics, it is typically restricted to research settings due to its reliance on script-based experimentation, manual parameter tuning, and technical expertise, creating a gap between advanced scientific modeling capabilities and accessible analytical tools. There is currently no integrated system that allows users to express high-level analytical goals in natural language while ensuring controlled, reproducible execution of complex epidemic simulation workflows. Therefore, our team will tackle this problem by designing and implementing an agentic AI-driven platform that orchestrates NIPA simulations and presents the results through an interactive web dashboard.

2.2 Risk analysis

Since this project will include an AI agent, there is a risk that the AI may misunderstand the user's goals, leading to incorrect simulations or parameter choices that do not match the intended analysis. What is more is that users may rely too heavily

on the AI-generated workflows without critically evaluating the assumptions or limitations of the underlying model, which may result in misuse or misinterpretation of the results.

Another important risk that we have to consider is the web dashboard's layout and interaction design. These components might not be suitable for all the users because of the differences in technical background, accessibility needs, or device types. Our design choices may also not be fully accessible for users with visual impairments or different interaction preferences, which could restrict participation and the interpretation of results.

As for some technical risks, the NIPA implementation may require significant computational time, especially for large datasets or repeated simulations. If the runtime is too slow, the web dashboard may become unresponsive, reducing usability and limiting interactive experimentation. In addition, since the agent can only operate through tools defined in the MCP server, incorrect, incomplete, or poorly specified tool definitions may restrict functionality or produce invalid workflows. Such issues may be difficult to detect immediately, especially if failures appear as model errors rather than tool errors. Fixing those issues would involve refining tool schemas, improving parameter validation, and implementing clearer error messages.

For integration, even if the AI agent, the MCP server, and the NIPA implementation work independently, failures may occur when connecting these components, since differences in expected input formats, data structures, or output handling may cause runtime errors or incorrect results. What is more is that, if unexpected inputs, invalid parameters, or runtime failures occur, insufficient error handling may lead to system crashes or unclear feedback to the user. Without robust validation and informative error messages, debugging may become challenging.

Lastly, as a general risk, the project may be affected by unforeseen issues such as team member illness, reduced availability, or uneven workload distribution, which can delay progress or reduce overall quality.

Chapter 3

Research objectives

This chapter presents the general objectives of our team, purpose, scope, and potential use of this project. Similarly, it describes the state of the art in agentic AI and scientific workflow automation, highlighting the limitations of existing systems and emphasizing the advantages and novelties of our project. Lastly, it identifies the primary stakeholders, users, and interested parties, clarifying who will benefit from the system and how it can be applied in research, education, and exploratory analysis.

3.1 General objective of this work

For this project, our team will make sure to have a working web-based dashboard with an agentic AI for epidemic modeling at the end of the module. In addition, we will have a completed design report with presentation slides and a poster to explain the details of our deliverables.

As we go through the project during this module, per week, we will have our own goals to achieve at the end of it. In case we cannot complete a set goal in time due to any circumstances, this will be immediately communicated on Discord, and we will try to come up with a solution as fast as possible.

3.2 Purpose, Scope, and Applicability

This section outlines the purpose, scope, and applicability of the proposed system. It defines the main objectives of this project and the potential users and contexts

in which the platform may be applied. Together, these elements clarify what this project aims to achieve, what aspects are included in the development, and how the resulting system can be used to support accessible and reproducible epidemic modeling.

3.2.1 Purpose

The purpose of this project is to design and implement an agentic AI-driven platform that enables accessible interaction with network-based epidemic modeling tools. In order to achieve this, the system will integrate an LLM and a web dashboard to allow users to define analytical goals in natural language and automatically translate them into reproducible NIPA simulation workflows.

3.2.2 Scope

As for the scope of this project, it includes the development of a modular system consisting of four layers: a Core layer containing the black-box NIPA implementation, a Service layer exposing the model through MCP-based tools, an Agent layer responsible for interpreting natural language goals and orchestrating workflows, and a Presentation layer providing a web-based interface for interaction and visualization. Our focus is on orchestrating the simulations and visualizing the results rather than modifying or improving the underlying NIPA algorithm itself.

3.2.3 Applicability

The proposed system is applicable to researchers, students, and analysts who wish to explore epidemic dynamics without requiring extensive programming or technical knowledge. By enabling natural language interaction with epidemic simulation workflows, the platform can support exploratory analysis, educational demonstrations, and reproducible research in epidemiology and network science. In addition, this project may serve as a general framework for integrating agentic AI with other scientific modeling systems.

3.3 State of the Art

Recent research has explored both scientific workflow automation and agentic AI systems that orchestrate complex computational tasks. In scientific domains, agentic frameworks such as SciToolAgent demonstrate how Large Language Model (LLM)-powered agents can integrate and automate diverse tools across research workflows, improving accessibility for non-experts while maintaining execution control (Ding et al. 2025).

Similarly, domain-specific agentic systems have been proposed in fields like computational chemistry (e.g., ChemGraph) where LLMs assist in designing workflows that combine simulation tools with natural language planning (Pham et al. 2025).

Studies evaluating the use of LLMs in scientific workflow development show that current models are effective at interpreting and explaining workflows but struggle with extending or modifying them in a reproducible way without substantial guidance and workflow controls (Sänger et al. 2024).

3.3.1 Limitations of other works

Most agentic systems are general and not tuned for complex epidemiological modeling. In addition, LLMs alone do not guarantee reproducible execution without structured tool interfaces and rigorous parameter control. Lastly, workflow tools often require technical expertise and lack natural interfaces, therefore creating a usability barrier.

3.3.2 Advantages/Novelties of our work

Our project focuses on an agentic AI specifically for network-based epidemic modeling (NIPA), bridging advanced modeling with accessible interaction. Since we will use a Model Context Protocol (MCP) service layer, it will ensure valid, repeatable tool invocations for simulations. At last, we will provide a web-based interface where users express goals in natural language and visualize the results.

3.4 Clients, Users, and Interested Parties

The primary stakeholders and clients of this project are our project supervisor, Mahboobeh Zangiabady, and the co-supervisor, Alberto García Robledo, who define the project objectives, provide guidance throughout the development process, and evaluate the final system. They represent the academic and research interests behind the project and ensure that the project meets the expected educational and technical goals.

Furthermore, the main users of the system are researchers, data analysts, and students who are interested in epidemic modeling and network analysis. Researchers in epidemiology or network science can use the platform to run epidemic simulations, explore inferred infection networks, and analyze outbreak scenarios. Likewise, data scientists and analysts may use the system to experiment with the epidemic datasets and evaluate predictive models. In addition, students studying fields such as data science, artificial intelligence, or epidemiology can use the platform as an educational tool to better understand epidemic dynamics and modeling techniques through an accessible interface.

Lastly, there are several interested parties who may benefit from the outcomes of this project. Universities and research institutions may use the platform as a demonstration of integrating agentic AI with scientific modeling workflows. Similarly, public health researchers may benefit from easier access to epidemic simulation tools for exploratory analysis or educational purposes. Finally, future developers or researchers may extend the platform by integrating additional epidemic models, improving the AI orchestration layer, or enhancing the visualization and interaction capabilities of the web dashboard.

Chapter 4

Requirements and Analysis

The following chapter outlines the project's requirements in a structured approach, beginning with requirements specification, followed by formulation and prioritization. Additionally, it then describes the stakeholder requirements and an analysis on the functional and non-functional requirements, ensuring that the system's objectives are clearly defined and translated into actionable specifications to guide the design, development, and validation of the AI-driven dashboard.

4.1 Requirements Specification

This section describes how our team approached the specification, formulation, and prioritization of requirements. It explains the use of Agile and Scrum to manage iterative development, the application of SMART criteria to ensure clarity and testability, and the MoSCoW method to prioritize requirements based on their importance and project goals.

4.1.1 Project Management Approaches for Requirement Specification

Requirements specification is the process of formally documenting analyzed requirements in a clear and usable way so they can be understood, reviewed, and adapted throughout the iterative development cycles. (Firesmith 2003). This is why our team used Agile and iterative development approaches to manage this project more easily. Agile emphasizes learning and flexibility over rigid plans (Dybå et al. 2014), thus allowing us to regularly review progress, incorporate feedback, and adjust

the priorities of the requirements. Additionally, our team used the Scrum framework because it applies Agile principles through defined roles, events, and artifacts, such as sprints and daily stand-ups through Discord calls. Scrum supports iterative development, frequent feedback, and adaptability, helping our team manage the complexity of integrating the project while ensuring continuous progress and alignment with our client needs (Srivastava et al. 2017).

4.1.2 Requirements Formulation

Our team used SMART criteria for our requirements because SMART makes requirements specific, measurable, achievable, relevant, and time-bound, which improves clarity and reduces ambiguity in what the system must do. By applying SMART principles to our project, they help transform our high-level ideas into clear, testable, and verifiable software requirements, ensuring that each requirement can be observed, measured, and validated during development and testing (Mannion & Keepence 1995).

4.1.3 Prioritization of Requirements

The SMART requirements will be prioritized based on the criticalness of their associated functionality to meeting the needs of our client and achieving the overall goals of the system. To achieve this prioritization, we applied the MoSCoW method, categorizing requirements as Must-Have, Should-Have, and Could-Have (Vijayakumar 2024). This approach allowed us to focus first on the requirements that are the most important for our system and the core purpose of the project, while providing flexibility to defer less essential requirements. Refer to Appendix A for the specified requirements and how they were categorized.

4.2 Stakeholder Requirements

In this component, we define the specific needs and expectations of the system's stakeholders that were previously mentioned in this section.

- As a user, I want to create an account using my email and password.

A user can sign up by providing an email and password, confirming the password, and having the credentials securely stored.

- As a user, I want to log in to the system.

  A user can access the dashboard using their registered email and password.

- As a user, I want to log out of the system.

  A user can end their session safely and ensure no further access until logging in again.

- As a user, I want to input analytical goals in natural language.

  A user can type high-level instructions that the AI agent interprets and translates into NIPA workflows.

- As a user, I want to visualize the simulation results.

  A user can view epidemic dynamics and inferred infection networks through the web dashboard.

- As a user, I want meaningful error messages in case of wrong inputs.

  A user is notified of input errors, invalid parameters, or system failures in a clear and actionable way.

- As a user, I want to see the AI agent's actions.

  A user can inspect workflow steps, tool calls, and parameters used by the agent.

- As a user, I want the dashboard to be easy to use without programming experience.

  A user can interact with the system intuitively, regardless of technical background.

- As a user, I want simulations to run efficiently.

  A user receives simulation results within reasonable time bounds for interactive analysis.

4.3 Requirements Analysis

In this section, we derive the functional and non-functional requirements of the system, specifying the features, behaviors, and quality attributes necessary to meet our goals for this project.

4.3.1 Functional Requirements

User accounts

- The system should allow users to create an account using only an email address and a password.
- The system must require a password confirmation to ensure accuracy.
- The system must allow users to log in using their registered email and password.
- The system must verify credentials and deny access if the email or password is incorrect.
- The system must hash the user's passwords before storing them in the database using a secure hashing algorithm.
- The system must allow users to log out.

Service Layer

- The system should validate all inputs before executing NIPA-related operations.

Agent Layer

- The system must accept high-level analytical goals expressed in natural language.
- The agent must translate user goals into structured, multi-step NIPA workflows.
- The agent must iteratively refine workflows based on results or user feedback.

- The agent must compare simulation strategies based on quantitative metrics such as peak infection value, total infected population, and mean squared prediction error.

Presentation Layer

- The system must provide a web-based chat interface for interacting with the agentic AI.
- The system must allow users to define analytical goals and constraints via the UI.
- The system must visualize the inferred infection network interactively.
- The system should allow users to inspect and reproduce previous simulation runs.
- The system must store chats.

4.3.2 Non-Functional Requirements

Usability

- The system should be usable by non-expert users with limited epidemiological modeling experience.
- The UI should prioritize clarity, interpretability, and transparency of agent behavior.
- The system should provide meaningful feedback and error messages to users.
- The system should provide clear messages when the user tries to log/sign in.

Performance

- The system should execute NIPA simulations within reasonable time bounds for interactive users.
- The system must complete login and account creation operations within 2 seconds under normal load conditions.

Reliability and Reproducibility

- The system should ensure that, for identical natural language inputs, it generates the same workflow and simulation outputs within a numerical tolerance of 1% for predicted infection probabilities.
- The system should log all agent actions, tool calls, and parameters used in simulations with timestamps for auditing and debugging.
- The system should handle invalid or missing inputs, returning descriptive error messages without crashing.

Security and Access Control

- The system should prevent unauthorized tool execution by the agent or external clients.
- The system must prevent brute-force attacks by limiting login attempts.

Transparency and Trust

- The system should make agent decisions and actions inspectable by users.

Chapter 5

Planning and Project Management

This chapter describes the methodology and project management approach used to develop the proposed system. It outlines the structured phases of the project, including planning, development, testing, and finalization of the deliverables. Additionally, it explains how our team organized tasks, monitored progress, and collaborated using tools such as a Gantt chart, Discord, and Trello. Finally, the chapter describes how communication with the project supervisor, who is also our client, and co-supervisor was maintained to ensure continuous feedback, alignment with project objectives, and resolution of potential issues.

5.1 Methodology

The project will follow a structured, phased methodology to ensure successful teamwork, technical correctness, and validation of the system.

- Phase 1: 2 weeks - Planning
- Phase 2: 4 weeks - Development
- Phase 3: 2 weeks - Testing
- Phase 4: 2 week - Wrapping up

5.1.1 Phase 1

Planning and communication are essential to completing the project successfully. Therefore, in this phase, we will focus on planning, documenting ourselves, and communicating with each other so that everybody is up to date with what they have to do next.

We are each going to communicate our daily plans and progress at the start and the end of the workday to let everybody know our plans and progress. Our main communication method will be through messages on Discord in a group chat, and we will hold meetings, either physically or through online calls, if necessary, to address problems or adjust our project-wide plan.

As a general responsibility, we will make sure that the tasks divided will be equal among us and based on our strengths to ensure that we can complete the project in time. We will make sure to identify the skills and technologies needed for those tasks so that each member understands what they are expected to do.

5.1.2 Phase 2

This phase will be the longest one since we will be focused on actually implementing the project while, at the same time, documenting what we do. We will divide the tasks according to what we have established in the first phase based on each of our strengths and capabilities.

During development, we will support each other by sharing knowledge, providing feedback, and assisting when challenges arise. Each of us will try to make a genuine effort to contribute positively to the group's progress and success. If any of us encounters difficulties during this phase, we will first make a reasonable effort to understand and solve the issue together. However, if further assistance is needed, we will document our efforts, including what we have tried, what resources we have consulted, and where we are encountering problems. This documentation will then be shared with our supervisor to see what we could do and how we could solve our issue to not further delay our work.

5.1.3 Phase 3

In this phase, we will focus on testing and further refining our implementation, if it is necessary, to make sure that at the end of the module, we will have a correct, working product. To ensure the system works as intended, this phase focuses on technical validation, such as unit testing, integration testing, reproducibility testing, performance testing, and user acceptance testing. These tests will allow us to detect errors early, validate correctness, and ensure the system meets the defined requirements.

By the end of this period, we should have a complete project that successfully allows users to express high-level analytical goals in natural language and have those goals translated into reproducible NIPA simulation workflows, using an AI agent and a web dashboard.

5.1.4 Phase 4

The last phase will focus on completing and refining all the deliverables for this module, such as finishing up the design report, making the presentation, and lastly, the poster. Our team will perform final checks on documentation and on the entire project to ensure that there are no errors left.

5.2 Project Management

To ensure that all team members remained aligned with the tasks defined during the planning phase, we created a Gantt chart to visualize the project timeline and track progress throughout the development process. This chart allowed us to monitor milestones, manage dependencies between tasks, and ensure that weekly goals were achievable within the available timeframe. Refer to Appendix B for visualizing the Gantt chart.

Furthermore, we kept ourselves updated through Discord messages or calls to let everybody know if a task had been successfully completed or in case anything went wrong so we could further resolve the problem together.

Lastly, to further support task management and organization, we also implemented a Trello board. Each team member could assign tasks to themselves, track their progress, and categorize them based on their current status, such as "To-Do",

"In progress", or "Completed".

5.3 Communication with the Client

As for maintaining effective communication with our supervisor, who is also our client, and with the project co-supervisor, we scheduled weekly online meetings on Microsoft Teams every Wednesday from 17:30 to 18:00. These meetings allowed us to present our progress, discuss completed tasks, and receive feedback on the current stage of the project. In addition, they also provided an opportunity to clarify requirements, address technical challenges, and ensure that our development remained aligned with the project objectives.

Besides these scheduled meetings, we maintained communication through email whenever urgent questions or issues arose that required immediate clarification. In this way, we were making sure that potential blockers could be resolved quickly and that the project could continue progressing without unnecessary delays. Regular communication with the supervisor and co-supervisor, therefore, played an important role in maintaining transparency, improving decision-making, and ensuring that the project met the expected academic and technical standards.

Chapter 6

Implementation and Detailed Design

This chapter provides a detailed overview of the design and implementation of the platform. It first describes the different design phases, including the initial concepts, mock-ups, and iterative improvements that led to the final version of the dashboard. The chapter then presents the final design of the platform, highlighting the main interface decisions, usability considerations, and visualization features.

In addition, this chapter explored the back-end implementation, explaining how the system components are structured and how the different layers interact with each other to support the platform's functionality. Together, these elements provide a comprehensive understanding of both the visual design and the technical foundations of the system.

6.1 Phases in Design

This section explains the design process followed during the development of the dashboard. It highlights the main design decisions that guided the creation of the interface, including the conceptual layout of the dashboard, the development of the project logo, and the creation of the initial mock-up. Additionally, this section presents the early front-end implementation and the design iterations that were made to improve usability and clarity, and later on, it describes the final design of the dashboard.

6.1.1 Design Choices for the Dashboard

Before developing the actual dashboard, the layout was first discussed and designed to provide an intuitive and user-friendly interface for interacting with the AI agent and exploring epidemic simulations. The main inspiration for this layout was the interface of modern conversational AI platforms such as ChatGPT, where users communicate with the system through a chat-based interaction. Since the core functionality of our platform involves users expressing analytical goals into natural language and receiving AI-generated workflows and results, a conversational interface was considered the most natural and accessible design choice.

6.1.2 Logo Creation

The design of our project logo that is illustrated in Figure 6.1 and Figure 6.2 combines elements of a virus symbol with network connections, reflecting the two main aspects of the system, which are the epidemic modeling and the network-based analysis. The central circular shape resembles a virus particle, while the surrounding nodes and connecting lines represent the interaction network between users.

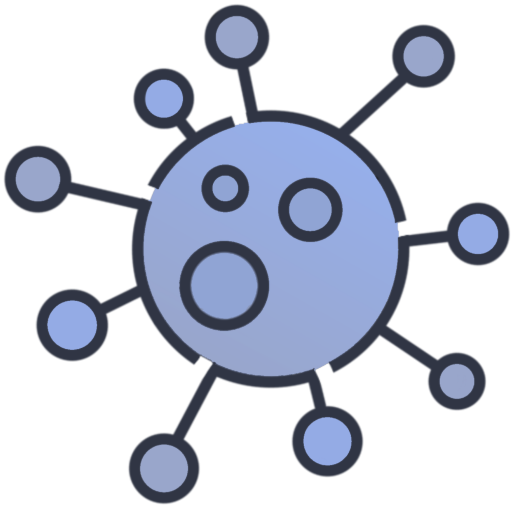

Figure 6.1: The logo of EpiNet

Figure 6.2: The logo with writing

By including connected nodes, it emphasizes how infections propagate through networks and how the system reconstructs these hidden relationships from epidemic data. This visual metaphor reflects the purpose of the dashboard, which is to analyze and simulate disease spread through inferred interaction networks.

Finally, the color blue was chosen intentionally for the logo, since it is commonly associated with technology, reliability, and scientific research, making it appropriate for a system that integrates artificial intelligence and data-driven modeling. In addition, blue conveys a sense of trust, clarity, and professionalism, which aligns with the goals of creating a transparent and accessible tool for epidemic analysis.

6.1.3 Mock-up Design

For the first mock-up, which was created using Canva, the layout was designed with simplicity and usability in mind so that it creates a clean and intuitive interface where the users could easily interact with the AI agent through a chat-based format without needing technical knowledge or navigating through complex menus.

Additionally, when visualizing the graphs, we came up with an idea to let the users switch their layout to either have the text and the graphs remain on one page, as presented in Figure 6.3, or to separate the text to be on the left part of the screen and the graphs to be on the right part of the screen, as shown in Figure 6.4, for better clarity.

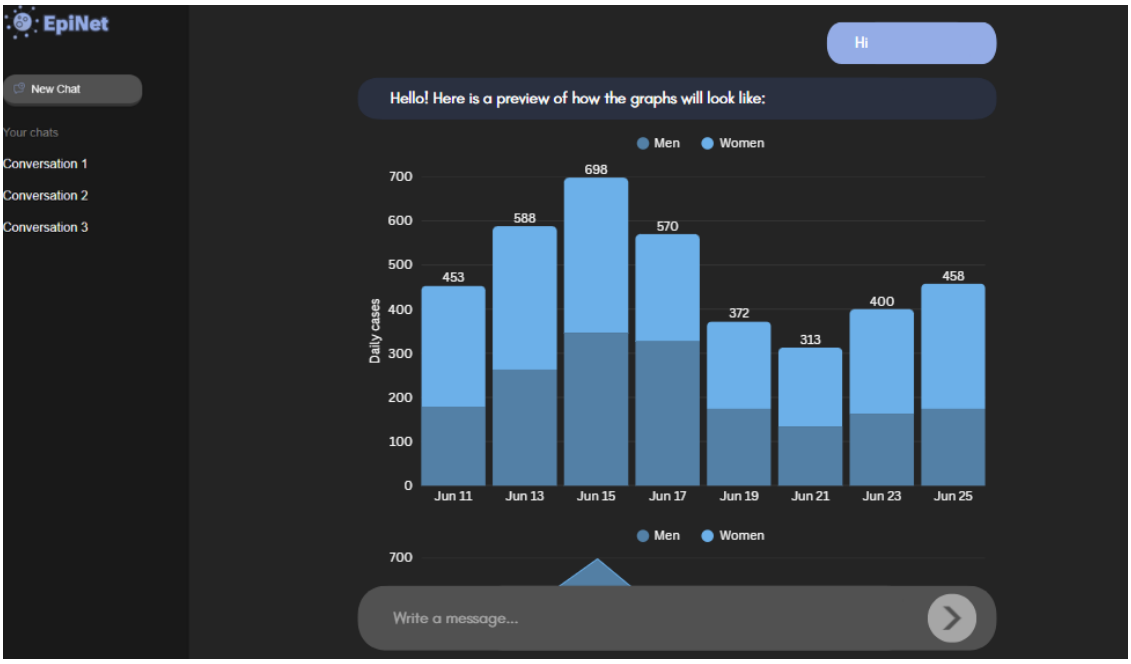

Figure 6.3: First choice for visualizing the graphs

Figure 6.4: Second choice for visualizing the graphs

Furthermore, the color scheme of the chat bubbles was also designed inten-

tionally to improve readability and usability and to match the logo's colors too. Messages from the user are displayed in light blue, while the AI responses appear in dark blue. This contrast allows users to easily distinguish between their inputs and the system's outputs, helping maintain clarity during longer conversations. Likewise, the graphs are also blue to match the color scheme of the dashboard.

Through this mock-up, the design aims to demonstrate a simple and familiar interface, similar to the ease of use offered by other modern conversational AI platforms, which helps us later in the future during the front-end development.

6.1.4 Design Iterations

The first implementation of the dashboard in the front-end, which is illustrated in Figure 6.5, closely followed the previously created mock-up, as it served as the main guideline for the interface design. The goal at this stage was to reproduce the overall layout and interaction model defined during the design phase, while beginning to integrate the core functionality of the system. By keeping the structure similar to the mock-up, we ensured that the interface remained simple, intuitive, and consistent with the original design intentions.

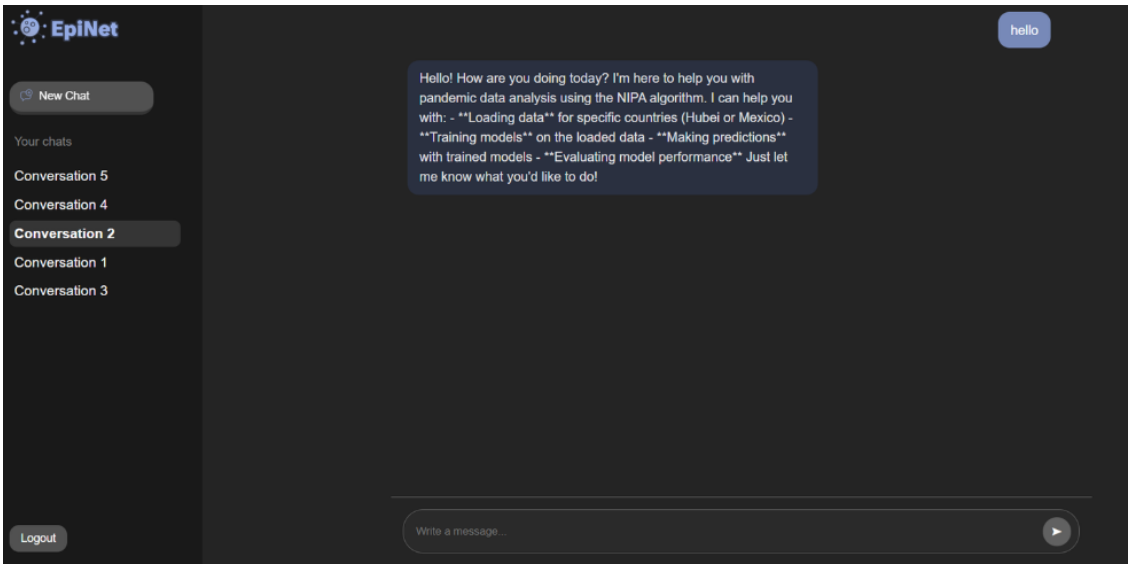

Figure 6.5: First implementation of the dashboard

However, during this initial implementation, we observed that the responses generated by the AI agent were not clearly structured, which sometimes made them difficult to read. As a result, improvements in formatting and presentation will be necessary to enhance clarity and readability. In the final iteration of the dashboard,

we plan to refine how the AI responses are displayed, for example, by organizing the content into clearer sections, improving spacing, and potentially highlighting key information to make the outputs easier for the users to understand.

Since this version represents only the first front-end implementation, the epidemic simulation graphs are not yet displayed. At this stage, additional back-end functionality still needs to be completed to generate and provide the data required for these visualizations. Once the back-end components are fully integrated, we will evaluate which types of graphs and visual representations are most suitable for presenting the simulation results in a clear and informative way.

In the second implementation, we focused on visualizing the results that the model returns on a map, since our client and co-supervisor suggested so in one of the weekly meetings our team had. The interactive map shows the prediction accuracy of the trained model across different regions in Mexico for each day of simulation, as shown in Figure 6.6.

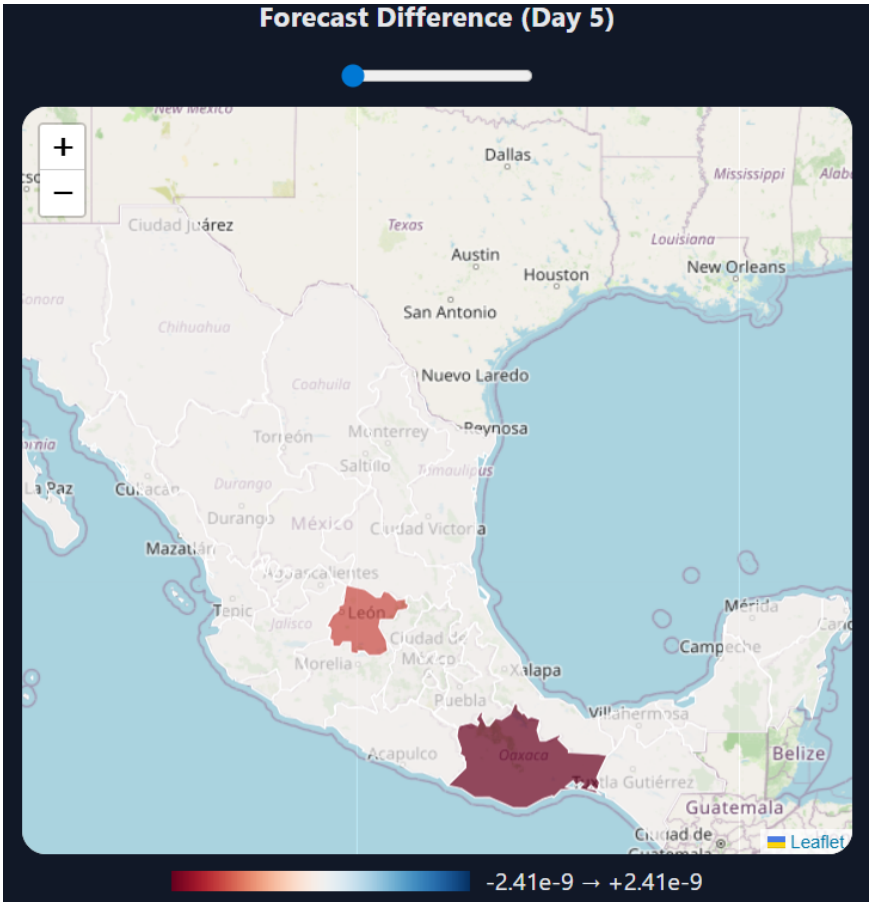

Figure 6.6: The interactive map across the regions of Mexico

A color scale is used to represent the difference between the predicted and ob-

served number of infections. Regions displayed in red indicate that the model underestimated the number of infections, while regions shown in blue indicate that the model overestimated the number of infections. The intensity of the color reflects the magnitude of the difference, where darker shades represent larger discrepancies between the predicted and observed values. Conversely, colors closer to white indicate smaller differences, suggesting higher prediction accuracy. A completely white region represents no difference between the predicted and observed values, which in many cases occurs when both values are zero, meaning no infections were observed or predicted in that region.

To provide more detailed insights, the map includes an interactive feature that allows users to hover over a specific region, as shown in Figure 6.7. Upon interaction, three key values are displayed, which are the observed value from the dataset, the predicted value generated by the model and the difference between those two values. These numerical outputs correspond directly to the results produced by the NIPA algorithm and are primarily intended for users with a research-oriented background who require detailed quantitative analysis.

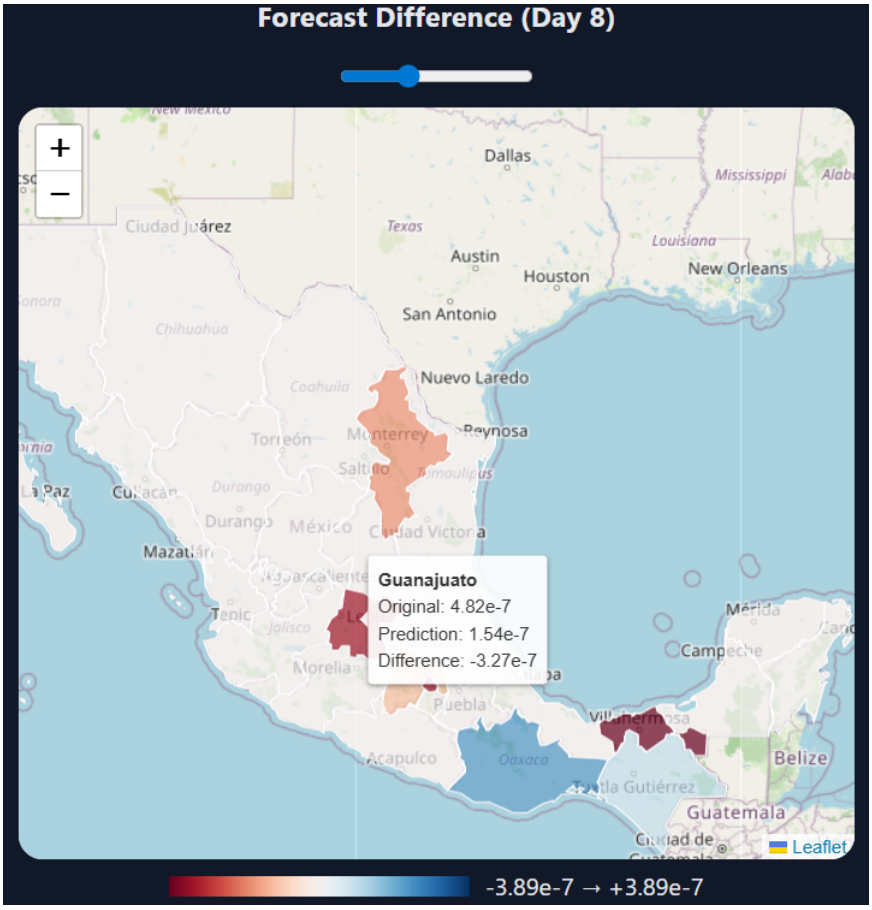

Figure 6.7: The interactive map visualizing the details about one region

6.1.5 Final Design

To provide a clear overview of how users interact with the platform, we have created a use case diagram, which is in Appendix C, that illustrates the main user interactions and summarizes the overall functionality of the system.

In the final design, before reaching the main dashboard, users are required to create an account by signing in, as shown in Figure 6.8 and then by logging in using their email address and chosen password, as presented in Figure 6.9. These authentications pages were designed to be simple and intuitive, following common design patterns used in modern applications. Additionally, the same color palette introduced in earlier design iterations was maintained to ensure visual consistency throughout the platform.

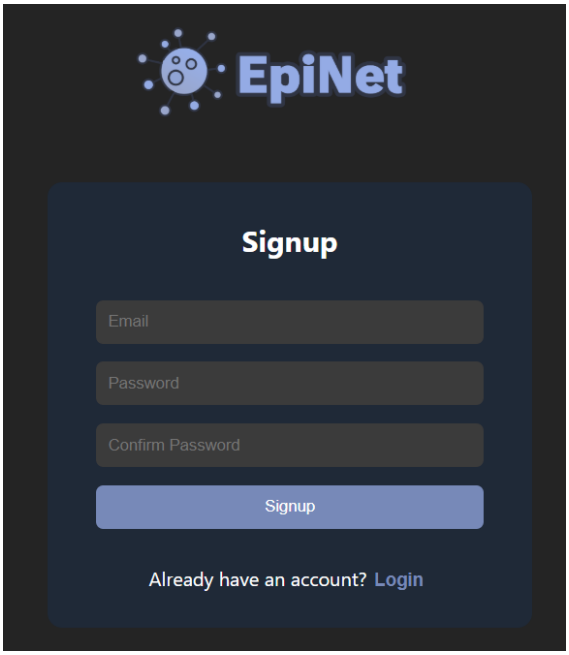

Figure 6.8: Sign-up page

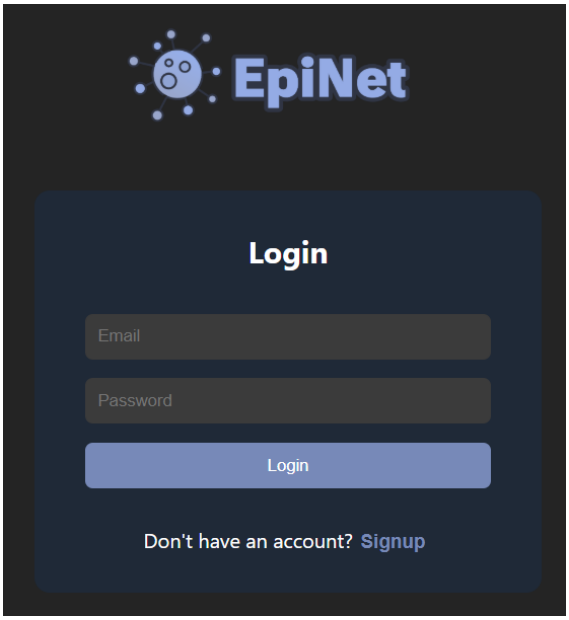

Figure 6.9: Login page

After logging in, users are directed to the main dashboard, as shown in Figure 6.10, where they are greeted with a welcome message explaining the purpose of the platform and guiding them on how to begin. Users are prompted to either create a new chat or select an existing one in order to start interacting with the AI agent. This design choice ensures that even first-time users can quickly understand how to use the system.

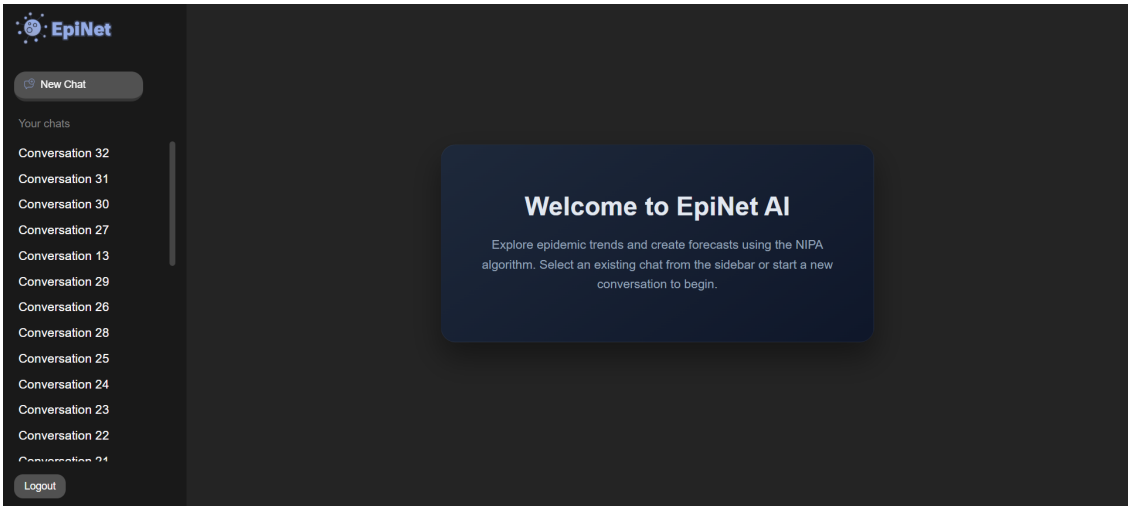

Figure 6.10: First page of EpiNet

Following user interaction with the AI agent, the dashboard interface presents a more refined and structured layout compared to earlier versions. The AI-generated responses have been improved in terms of formatting and organization, making

them easier to read and interpret, as is presented in Figure 6.11. This enhancement contributes to a more intuitive and user-friendly experience since it is easier to follow and read the responses now.

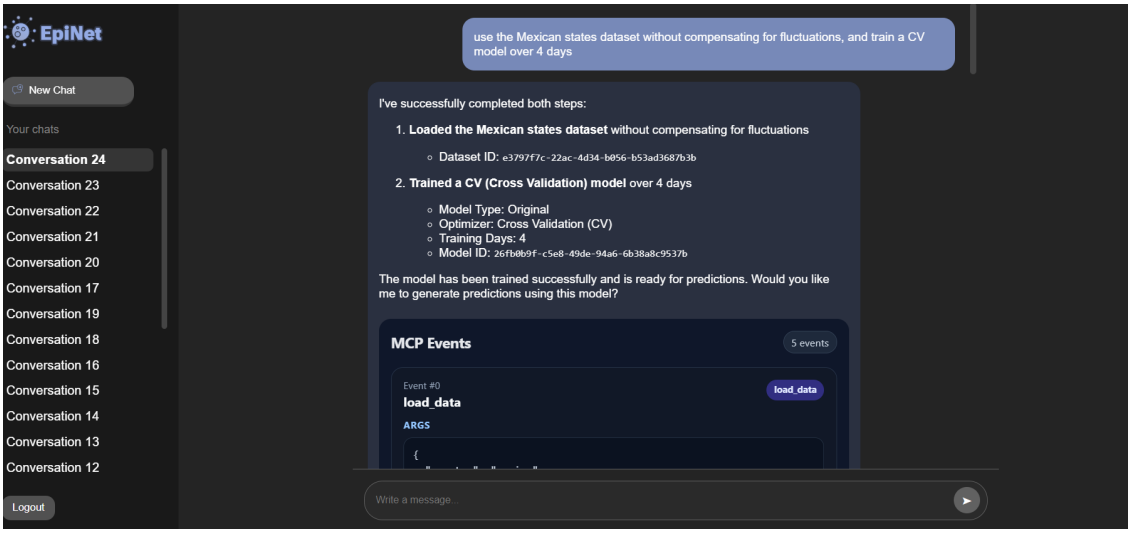

Figure 6.11: Final design of the dashboard

In addition to improvements in the chat interface, the final design also includes other advanced visualizations than previously seen in the other design versions. One such feature is another interactive map of states in Mexico that represents the relationships between regions based on the inferred network model, as shown in Figure 6.12. This visualization illustrates how different regions influence one another in terms of infection probability, providing users with deeper insights into the spread of diseases and the underlying network dynamics.

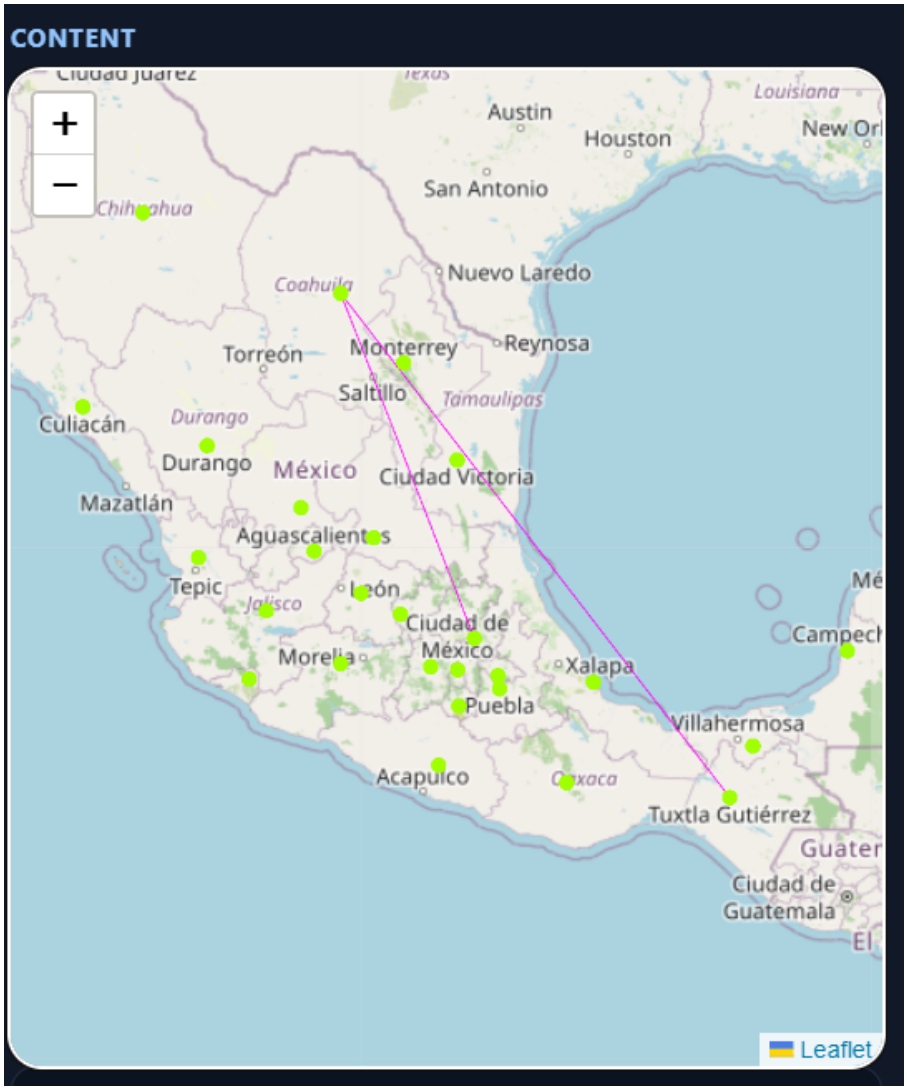

Figure 6.12: Map that shows how each state are influenced between them

Finally, our team included a graph that presents the progression of infections over time across all of Mexico as shown in Figure 6.13. The Original columns represent the proportion of the infected population based on observed data for each day, expressed as a decimal. In contrast, the Prediction columns represent the same metric but are derived from the outputs of the trained model using the NIPA algorithm.

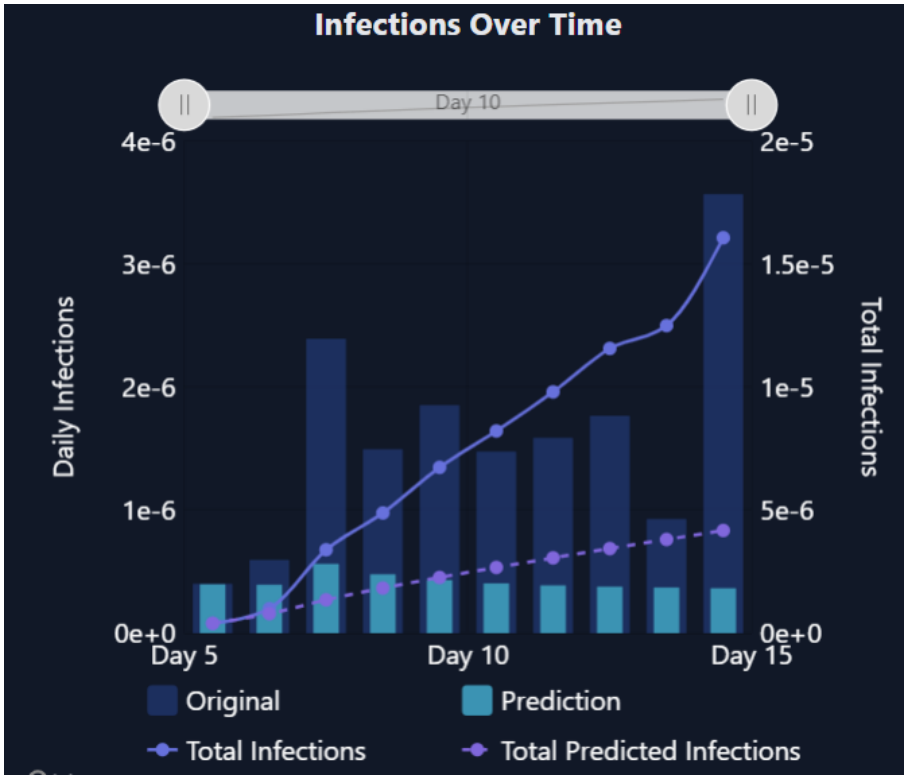

Figure 6.13: Graph that shows the infections over time in all of Mexico

Furthermore, the Total Infections line illustrates the cumulative change in the proportion of the infected population over time, based on the observed data. Similarly, the Total Predicted Infections line represents the cumulative proportion derived from the model's predictions. This comparison allows users to assess how closely the model's predictions align with the actual observed trends.

6.2 Overview of Backend Architecture

The backend is designed as a microservice-based architecture supporting modularity and clean separation of concerns, where services handle various machine learning processing, orchestration, and LLM interaction. A detailed overview of how the system is structured and how it works can be seen in Appendix D. This design allows the application to be scalable and maintainable and allows for an efficient team workflow, where division is clear. Furthermore, computationally intensive tasks such as model training are handled efficiently through this design, and each service can have differing environments best suited for their tasks. The system consists of three core backend services: the ML service, the MCP service, and the orchestration server. These services also interact with a cloud-hosted LLM as

well as a database for persistence. These services communicate with one another through RESTful APIs.

The ML service is responsible for executing the NIPA algorithm and its various stages, including loading data, training, prediction, evaluation, and more. Furthermore, this service is responsible for handling concurrent work through an asynchronous task queue system.

The MCP service connects the ML service with an LLM through wrapping its functionality to LLM-understandable tools. Additionally, this service handles complex tool-calling workflows where an LLM can, in one user prompt, call multiple tool calls both synchronously and in parallel. Lastly, this service provides an entry point for user prompts where multiple concurrent users can have their prompts processed simultaneously.

The orchestration server manages the coordination of these services, as well as handling user authentication and database access.

An example interaction can be described as follows: a user fills in and submits the prompt through the React client application, which is next sent to the Express backend (orchestration server). The backend authenticates, stores the prompt, and calls on the MCP service to further process the prompt. The MCP lets the LLM know what tools are available and how they should be used and then allows the LLM to call those tools based on the prompt. The LLM may call some tools that invoke tasks on the ML service where various phases of the ML pipeline may be computed, which upon completion will be fed back to the LLM in the MCP service. When the LLM has used all the tools it needed and those tools have finished their execution, the agent response as well as the tool calls and their inputs and outputs are returned to the orchestration layer, where they are stored and sent to the user.

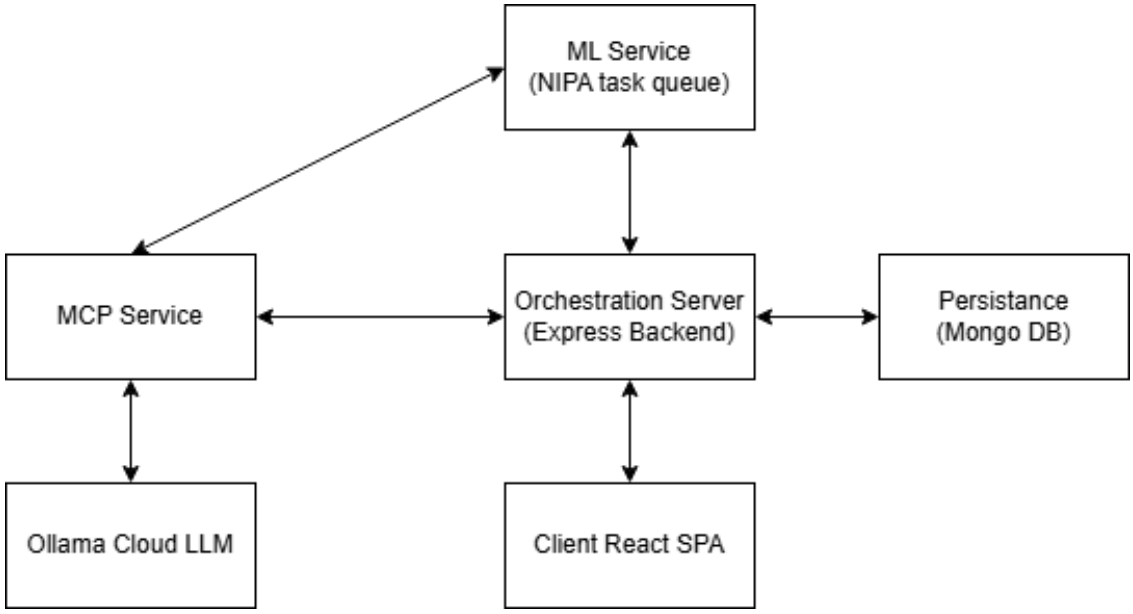

Figure 6.14: A diagram illustrating the overall backend architecture, showing service interaction

6.3 Architectural Design Decisions

This section discusses the main architectural design decisions and their reasoning. The challenges that need to be addressed are the high computationally intensive machine learning tasks and the need for a modular, maintainable, and scalable system.

6.3.1 Microservice Architecture

One of the core design decisions was to structure the backend into multiple services, adopting the microservice architecture rather than a single monolithic application. Three main components were devised: the ML, MCP, and orchestration services. Each service has their own responsibilities with minimal overlap.

The machine learning requirements are very different from the rest of the application; therefore, it benefits from its own service to drastically reduce complexity. Furthermore, this service requires a specialized environment with specific dependencies and a slightly outdated Python version, making microservices more desirable due to ease of isolation. Due to the machine learning tasks being the most computationally intensive part of the application, it also needs to address real efficiency requirements. These requirements need parallelism to operate efficiently when mul-

tiple such tasks are needed. The culmination of all the complexity from this service makes microservice-based architecture desirable by keeping it separate from the rest of the system.

Scalability is another challenge, which is addressed through the microservice design choice. Since these services operate independently, they can be scaled as such based on the system bottleneck. As discussed before, the machine learning operations are the main bottleneck of this application; therefore, it can be scaled without needing to scale the MCP and orchestration services. This argument also supports maintainability, as each service may be rewritten or slightly changed without affecting the rest of the application as long as the API contracts are kept.

Overall a microservice-based architecture supports a scalable, maintainable, and modular application, solving important challenges the application needs to overcome.

6.3.2 Stateless Pipeline Design

A second major decision was to design the machine learning pipeline to be stateless and coordinated through shared run metadata. When designing machine learning experiments, reproducibility is very important, which this design decision supports.

Each stage of the machine learning pipeline can be invoked without in-memory data maintained across stages. Instead, each task accesses a run file containing configuration and previously completed operations the current task might need. Although each stage is independent from an in-memory perspective, they still depend on each other to create meaningful output; for example, to predict, it is necessary to provide a model that must have been trained on data.

An important benefit of this stateless design is that it gives the analyst performing experiments a large amount of freedom. This is due to the system allowing any stage of the pipeline to be called, not locking into a singular workflow, allowing for multiple variations of a machine learning stage to be executed for comparison and defining a different workflow for each. Furthermore, this design is well aligned for remote invocation where other services or clients do not need special handling.

6.3.3 Asynchronous Task Processing

A major problem needing to be solved is the computationally intensive and long-running machine learning tasks. To solve this, the design decision to delegate long-running tasks to an asynchronous task queue was made. The various machine learning pipeline tasks are executed asynchronously using Celery and Redis (libraries used in Python). This allows long-running jobs to not block the application. Multiple requests can be handled concurrently both from different users and from the same user. The application remains responsive while handling intensive tasks efficiently.

6.4 ML Service

This section describes the machine learning (ML) service of the platform, explaining its purpose and how it works.

6.4.1 Purpose and Responsibilities

The ML service is responsible for executing machine learning tasks and keeping track of experiment workflows using the NIPA algorithm and exposing these functionalities through RESTful API endpoints. This is the most computationally demanding part of the system handling data loading, training, and inference.

This layer benefits from separation by isolating computationally intensive operations from user-facing backend components. This allows this layer to scale independently while handling concurrent work. Furthermore, this service focuses on configurable and reproducible experiment execution while separating tasks to allow users to inspect intermediate results and decide whether to proceed, adjust parameters, or change the workflow.

6.4.2 Pipeline Architecture

This section describes the pipeline architecture necessary for interacting with the NIPA algorithm. The pipeline design allows individual stages of the machine learning workflow to be invoked independently. These pipeline stages include operations such as configuration loading, model training, evaluation, and inference.

Alternatively, the interaction with the NIPA algorithm is done through a tightly coupled sequential process. This design is rejected due to the required user expertise, limited user freedom through disabling intermediate result decision-making, and supporting inefficient computational handling. This design is computationally inefficient as it supports completing the entire sequence upon each change, whereas less computation could be achieved if the user were to change only one stage. Furthermore, when needing an experiment with controlled stages, but with selected variability, this design would require the sequence to be computed as many times as the amount of variability. These problems are solved by a stateless pipeline architecture. A stateless architecture allows any stage to be invoked and documented. Upon a user-requested change, only the stages with the change are required to recompute, drastically improving efficiency. A controlled experiment is also more computationally efficient, as only the stage under variability is recomputed multiple times. Against conventional expectations, where invocation of a sequential set of tasks requires less expertise from the user without necessitating knowledge about execution order, our system delegates this to the LLM, allowing users to benefit from independent invocation without ordering knowledge.

6.4.3 Experiment Configuration

To support the stateless pipeline architecture, a run configuration approach is chosen. Instead of requiring all parameters needed for execution logic on invocation, configuration files specify runtime settings such as dataset selection, model parameters, and pipeline behavior. This configuration system allows stages to be linked with one another, without relying on in-memory data tracking.

Each pipeline execution, or experiment, has its own run data and configuration. This configuration documents each stage, what parameters were used, which prior state it depends on, and any intermediate output produced. Through this design multiple experiments can be set up independently, and for each experiment the execution context is kept.

Configuration access and storage is implemented through file system interaction rather than a database. This decision is made due to the simple usage pattern, not requiring the overhead a database would introduce. There is no complex querying needed, as only one run file is needed at a time, and each run file has a low amount

of data to parse through. Furthermore, concurrent access is handled simply through atomic commits and sparse locking.

6.4.4 Concurrency

As described previously, machine learning tasks in our system are computationally intensive and time-consuming. To continue processing incoming requests and ensure efficient processing, the service delegates intensive tasks to workers.

This section will describe how work is orchestrated and delegated in the ML service. First our application submits machine learning jobs as per the API requests through the Celery orchestrator to the Redis message broker. The celery workers then claim the jobs from the broker and execute them. Each worker, once a task is claimed, further delegates the work to its child processes. After task completion the status in the message broker is updated, and the output is saved back to Redis, which the Celery orchestrator retrieves upon API request. Furthermore, Celery also handles retries for increased system robustness, necessitating an idempotent design of Celery tasks.

Celery tasks belonging to the same run need to process shared resources, mainly the run file, and any artifact two processes may be using (artifacts refer to intermediate results such as models, predictions, and loaded data). Since only tasks sharing a run may access the same resources, a run-level lock is implemented. The lock is implemented using Redis, allowing multiple workers, possibly on different machines, to protect critical code accessing run-related shared resources. Differing from operating system file-level locks, which would work only on the same host machine. To scale workers on multiple machines, either complex storage syncing is required or the better strategy would be to introduce a load balancer with run-level hashing, ensuring the same run-related tasks always run on the same machine. The run-level lock is used during critical sections but released on computationally intensive tasks such as model training to allow parallelism where it's necessary.

This section will describe the locking pattern used in the celery tasks. First, retrieval of all related data from the run file is done under lock. Next, given the data now loaded in memory, the intensive task is completed outside a lock. Finally, the output is saved atomically, and the run file is updated again under lock. To ensure idempotency, the run file includes operation states, which are updated and

read inside the locks, which signal how far previous retries have gotten and how far the current line of execution is. In case of partial prior execution, some output may be reused within retries.

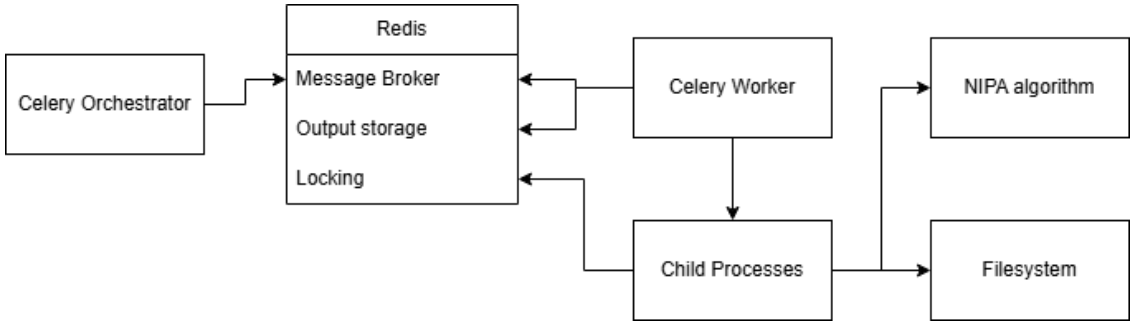

Figure 6.15: A diagram illustrating ML service architecture

6.4.5 API design

The ML service exposes its functionality through a set of REST API endpoints. It allows the creation of new experiments or runs, returning with a new unique run identifier for which the caller is responsible for storing. The run identifier is used for all subsequent tasks with the same execution context. Similarly, when pushing a task for processing, a task identifier is returned. To get the result of a task, the caller must poll with the task identifier, and if the task has finished, an output is returned; otherwise only the state of the task is returned. The tasks that are exposed are the various stages of the pipeline execution, such as loading data, training, prediction, and evaluation. Additionally, some endpoints for data retrieval and visualization are exposed.

This interface allows the ML, MCP, and Orchestrator services to be independent. The MCP service is the main consumer of this API, where it translates requests from the LLM into interactions with this interface. The role of the MCP service is described in more detail in the following section.

6.5 MCP Service

This section describes the Model Context Protocol (MCP) service, explaining how it enables structured communication between the AI agent and the underlying system components.

6.5.1 Purpose

The MCP service is responsible for integrating the orchestration backend, the ML service, and the cloud-hosted LLM. Primarily its responsibility is translating the functionality of the ML service into structured LLM-callable tools. Furthermore, it processes user prompts in addition to history and forwards them to the LLM.

The MCP service is also responsible for managing the execution flow of LLM interactions with the exposed toolset. A continuous workflow is maintained between the LLM and the pipeline tools, allowing the system to support parallel tool calls and sequential execution. Sequential execution is referred to here as the execution flow where operations depend on the results of previous tool calls. Consequently, the LLM is capable of dynamically coordinating a multi-step analytical process.

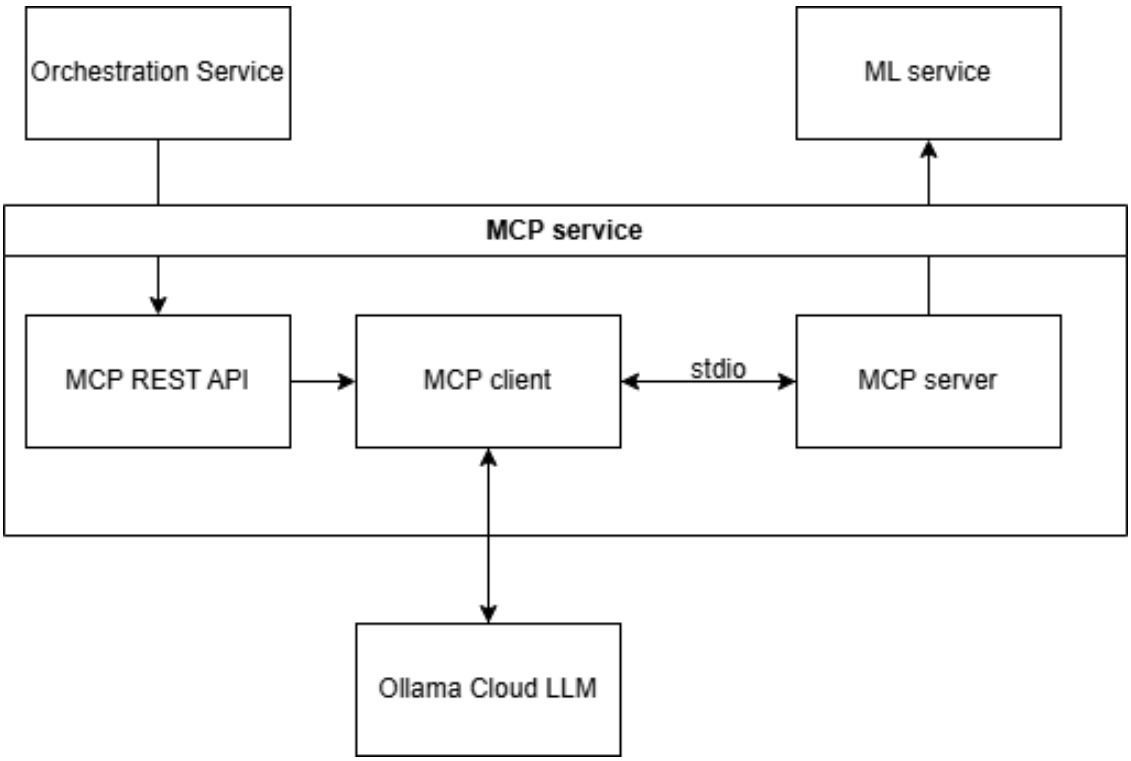

Figure 6.16: A diagram illustrating the overall backend architecture, showing service interaction

6.5.2 MCP Server

The MCP server is responsible for wrapping the ML service exposed REST endpoints and transforming them into LLM-callable tools. This is implemented through the FastMCP library.

The exposed tools map to the main stages of the machine learning pipeline, as

described in previous sections. To help the LLM manage long-running asynchronous operations, the server exposes two tools: the poll and poll wait. Delegating continuous polling to the LLM is wasteful and error-prone, which is why an additional continuous backoff-based polling mechanism is implemented at this layer, allowing the LLM to retrieve the result in one long-lived tool call. Importantly, this continuous polling logic is non-blocking, allowing for other interactions with the MCP server.

The MCP protocol comprises its tool signature through function name, parameters, output, and description. The description of each tool is comprehensive, describing the purpose, use, parameters, prerequisites, and dependencies.

A key design decision in the MCP server is the use of context injection in the tools, where context is given to the tools that the LLM is not responsible for. As previously documented, experiments are associated together with a run identifier, which through this design alleviates the responsibility of the LLM by abstracting it away through context injection. This also forces the LLM to stay operating within the given experiment specified by the prompt request.

The MCP server remains lightweight as all machine learning logic is delegated to the ML service, while it acts only as a thin layer exposing tool calls.

6.5.3 MCP Client

In addition to the MCP server, this service also includes an MCP client. While the MCP server exposes LLM-callable tools, the MCP client manages a connection with the LLM and is responsible for managing its workflow with the given tools. Furthermore, it takes in information from the orchestration server originating requests to prompt the LLM. Sequential and parallel tool calls are also handled at this layer. The MCP client is implemented by using the Pydantic AI library, handling complex work execution.

The LLM is first given the user prompt and prior chat history, including all user messages, agent responses, and tool calls from a chat. The LLM then can decide to execute tools, doing so sequentially in parallel or a mixture of both. After the LLM finishes execution of the tools, this service documents the agent text response as well as all the tool calls made, including the LLM-used parameters and tool call outputs.

The design decision to give the LLM the entire chat history is supported mainly by the large context windows modern LLMs support. For this use case the chat history practically never grows beyond a context window the LLM can handle, especially as chats are scoped as self-contained experiments. However, if this were to become a problem or token usage became expensive, a combination of strategies can be used. One such strategy is the most recent sliding window strategy, where only the k most recent user messages and agent responses are kept. Another strategy is to summarize with the LLM blocks of previous messages, reducing the context size. Additionally, some LLM providers support internal caching mechanisms such as key-value caching (KV caching), reducing repeated computation.

6.5.4 Lifecycle Management

An important design decision was to run the MCP server instance as a persistent subprocess, present throughout the application's lifetime.

Alternatively to a long-lived subprocess, a new subprocess can be initialized on each request. This lifecycle strategy is rejected due to the increased overhead, multiple initializations, and the resulting decreased responsiveness.

The MCP server subprocess is run as an stdio server, allowing the MCP client to interface with the LLM and the server through the standard input and output of the spawned child process. This communication model simplifies development by eliminating the need for additional network configuration.

Furthermore, since all tool calls are stateless and non-blocking, many agents can interface with this server at the same time, supporting multiple concurrent interactions. This allows the MCP layer to be interacted with by multiple users, as well as multiple parallel interactions by the same user.

6.5.5 API Design

The MCP service exposes one REST API endpoint, which the orchestration server consumes. The endpoint exposes an input for user prompts, chat history, and the run identifier. The user prompt and chat history are forwarded to the LLM, while the run identifier is propagated to the tool calls.

If the LLM decides to use tools and a complex workflow, it is further handled by the MCP server and client, respectively. On execution completion the agent

response and a list of tool calls, including their inputs and outputs, are sent back to the orchestration server for further handling.

6.6 Orchestration Service

This section describes the orchestration service, explaining how it coordinates the interaction between the AI agent, the MCP service, and the underlying system components to execute simulation workflows.

6.6.1 Responsibilities

The orchestration service is responsible for serving users' requests using the other microservices, which the users do not directly interact with. The orchestration server is the public-facing API and interface that connects potential user interfaces/clients/users to the use cases and functionality of the app. In terms of a "Model-View-ViewModel" architecture, the orchestration server would be the ViewModel.

This choice was made so that the development of the frontend of the application is straightforward, clean, and modular so that different microservices or backend implementations can be implemented without breaking changes for the frontend (instead, the orchestration server is responsible for orchestrating the available internal microservices to behave as the external interface).

Its responsibilities are:

- Authentication for the users
- Creating, viewing chats, and sending messages in a chat
- Ensuring users' data is persisted
- Sending messages and receiving agent messages back using the microservices

6.6.2 Authentication

The orchestration server is designed as a REST API. Therefore, authentication is implemented via "sessions," particularly server-side sessions with the client's

opaque session identifier stored in HTTP-only cookies with strict CORS (Cross-Origin Resource Sharing). This prevents stealing credentials via JavaScript, as the cookie is not accessible by JavaScript. The CORS aspect ensures that the credentials are only issued for requests from the application's domain, preventing CSRF (Cross-Site Request Forgery) on external malicious sites. A risk we still have to watch our for in developing and maintaining the application is possible code injection into the browser, which could perform malicious requests. However, we have sanitized our user input and encourage watching out for this risk.

The reason why we designed authentication this way is to prevent credential theft and CSRF with a simple design. We could have decided to stick the client's session identifier in the header if our app would not run in the browser, but because we wanted to deploy our app in the browser, we wanted to adapt to its security features and requirements.

We did not design authentication around JWTs (JSON Web Tokens) because we had no reason to prematurely optimize scalability and server load at the cost of a more complex implementation to be as secure.

Users must have a valid session identifier for any request except logging in, signing up and logging in.

6.6.3 Chat Management

Users' chats are persisted in the database. The orchestration server exposes functionality to create chats and list the chats of a user in the order of last access. The messages in a chat can be requested in the order of the conversation, including both messages from the user and the agent. Agents' messages have an MCP tool call list associated with them, which is also retrieved. Based on this information, the frontend or other potential consumers of the orchestration server can show the chat in a user-friendly manner and display different visualization widgets based on what tools the agent used.

When creating a chat, the orchestration server additionally creates a "run" on the ML service so that the tools the agent will call for the chat are sandboxed. The association of the run with the newly created chat is persisted to the database by the orchestration server.

When a user sends a request to send a message, the orchestration server persists

the user's message to the database, and makes a request to the MCP service, aggregating the chat's entire message history with the users' new message. The MCP service responds with the new message from the agent and the list of tools it called. This is then persisted to the database, and the orchestration server sends the user the message from the agent and tools it used.

Unauthenticated users cannot do anything related to chats. Users can only access chats that they created, so users cannot access each other's chats.

Chapter 7

Testing

This chapter describes the testing process of the system, focusing on how the different components were validated to ensure correct functionality and reliability. It outlines the testing approaches used for each service and explains how key features were verified through unit and integration testing.

7.1 ML Service

The ML service REST API was tested using unit tests implemented with Python's built-in unit test module. The testing strategy included both short-lived tasks to verify basic functionality and a longer end-to-end workflow test, which requires more time due to the execution of computationally intensive tasks.

The following aspects were tested:

- Successful creation of a run
- Retrieval of an existing run
- Handling of requests for non-existent runs
- Initiation of data loading tasks
- Tests the full workflow for creating a run, loading data, training a model, making a prediction and evaluating the prediction

These tests focus on verifying the behavior and reliability of the API, rather than the correctness of the underlying NIPA algorithm.

7.2 MCP Service

The MCP service REST API was also verified using unit tests within Python's build-in unit test module. These tests focus on verifying that the correct tools are invoked based on example user prompts issued to the agent. This ensures that the agent can correctly map user requests to the appropriate MCP tools. Additionally, this testing approach primarily evaluates the interaction between the LLM and the available tools, rather than the correctness of the tool outputs themselves.

The following functionalities are currently tested:

- Loading data
- Training a model
- Generating predictions using a trained model
- Evaluating prediction results
- Polling the status of a task
- Retrieving a model's network

7.3 Orchestration Server

The orchestration service REST API was tested as the other services using the already build-in unit tests in Python. These tests focus on verifying the correct behavior of core user interactions, including authentication, authorization, and chat functionality. Furthermore, the testing suite evaluates whether the most common user interactions behave as expected.

The following functionalities are currently tested:

- Successful account registration with an unused email
- Rejection of registration attempts with an already used email
- Failure of registration with invalid or incomplete credentials
- Successful login with valid account credentials
- Failure of login with incorrect password

- Failure of login with an unregistered email
- Enforcement of authentication for protected routes (accessible only after login and inaccessible after logout)
- Successful creation of chats
- Retrieval of existing chats
- Restriction of chat access to the owning user
- Restriction of message access to the user's own chats
- Successful sending of messages with corresponding LLM responses
- Proper handling of attempts to send messages to invalid chats
- Retrieval of messages from valid chats

Chapter 8

Conclusions

In this chapter, an overview of the development, management, team responsibilities, requirements evaluation, limitations, and future improvements of the EpiNet project was presented. It describes how the Software Development Lifecycle (SDCL) guided the project and how team responsibilities were divided based on individual strengths while maintaining collaboration for the key tasks. Furthermore, the chapter addressed the evaluation of the system requirements, together with the limitations of the system. Finally, it discussed possible future implementations that could improve the overall project.

8.1 Software Development Lifecycle

The development of this project followed the main stages of the Software Development Lifecycle (SDLC), as shown in Figure 8.1, to help organize the complex software project and improve software quality through structured development and continuous refinement (Ruparelia 2010). Our team began with identifying and gathering requirements, where stakeholders' needs and system objectives were defined, followed by the planning phase, during which the project timeline, responsibilities, and development strategy were established. Next, the design phase focused on defining the system architecture, dashboard layout, and interaction mechanisms between the AI agent and the underlying modeling tools. The system was then implemented, where the different layers of the architecture and the dashboard interface were developed. After implementation, the system underwent testing to validate functionality, performance, and reliability. Once the system reached a sta-

ble state, it could be deployed to allow users to interact with the AI-driven epidemic modeling platform. Finally, the maintenance phase ensures that the system can be improved over time by fixing issues, refining features, and adapting the platform to future requirements.

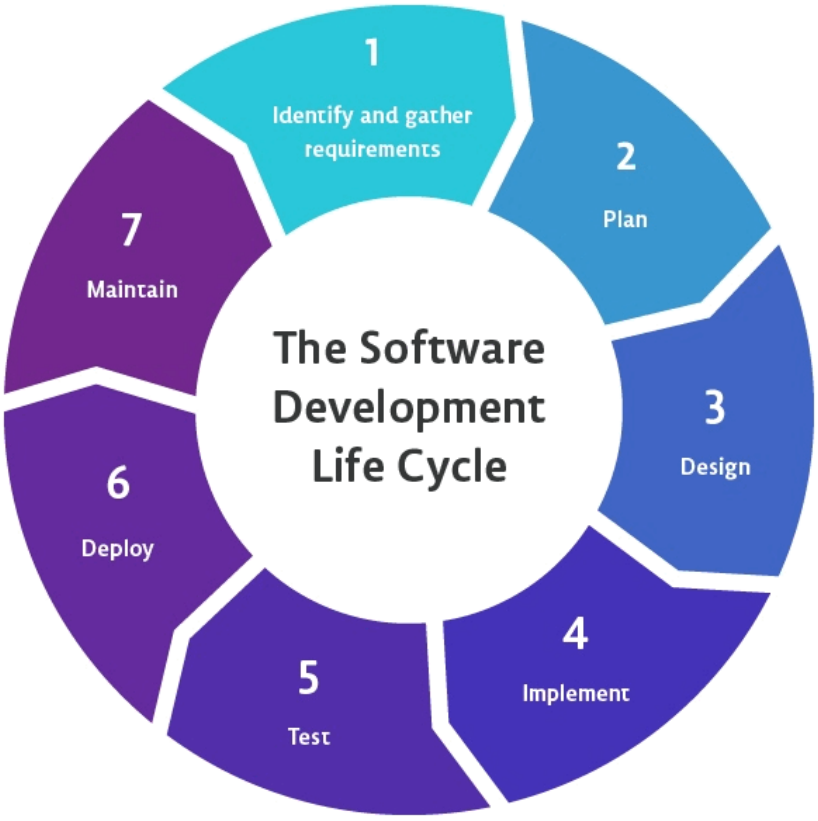

Figure 8.1: The Software Development Lifecycle (Source: (PVS-Studio 2023))

8.2 Team Responsibilities

Responsibilities within the team were divided based on each member's interests and strengths to ensure efficient progress and high-quality results. Besides individual responsibilities, some of them, such as design decisions, testing, implementation, and documenting, were handled as a group effort to ensure consistency and shared understanding. By doing so, this approach allowed us to work both independently and collaboratively throughout the project. Below, the responsibilities of each team member are listed:

- **Daria:** Lead writer of the report, front-end developer, and main designer of the diagrams, logo, and dashboard.

- **Konstantinos:** Main developer in the front-end, helped connect the front-end with the back-end and create diagrams.
- **Daniel:** Main Express back-end developer, connected the front-end with the back-end and helped create the machine learning service.
- **Gela:** Main developer in the back-end for the machine learning service, helped in the Express development and responsible for developing the MCP service.
- **Ashkan:** Main developer for the Model Context Protocol server and the AI model and helped in the Express development.

8.3 Requirements Evaluation

Regarding the defined requirements, the team successfully implemented the majority of them, with only a few less critical requirements remaining incomplete. The primary focus was placed on the core functionalities of the system, ensuring that the platform operates correctly and meets its main objectives that our client wanted.

The first incomplete requirement concerns the prevention of brute-force attacks:

- *The system must prevent brute-force attacks by limiting login attempts.*

While this is an important security feature, it was considered lower priority within the scope of this project, as the main focus was on the AI agent and the dashboard. Nevertheless, this requirement should be addressed in future work to improve the overall security of the platform.

The second requirement that was not fully completed was related to logging all the agent details:

- *The system should log all agent actions, tool calls, and parameters used in simulations with timestamps for auditing and debugging.*

This functionality has been largely implemented, but the inclusion of timestamps is still missing. Adding timestamps would further enhance traceability and support more effective debugging and analysis.

Overall, the project successfully fulfills the most essential requirements, delivering a functional system that meets its primary goals. The remaining requirements

represent opportunities for future improvements rather than critical limitations of the current implementation.

8.4 Limitations and Future Improvements of the Project

This section discusses the main limitations of the current system and identifies areas for future improvements. It highlights aspects that were not fully implemented or could be enhanced and outlines potential directions for extending the functionality and performance of the platform.

8.4.1 Scalability to Different Countries

Currently, the project is hardcoded and only supports a dataset from Mexico created during COVID times Government of Mexico (2021), more particularly, the dataset per state. This is due to a propagated limitation present in the original NIPA implementation. The original NIPA implementation does not load datasets by arbitrary CSV file paths, but it loads a dataset by names defined in the source code, looking for a specific file path and applying a particular transform to normalize the column names and values. We have not changed the original NIPA implementation to load CSV files by file path. Due to this limitation, scalability to different countries is not possible via using the web interface but requires editing the source code across all layers of the project. Here is a summary of the changes that are needed to allow users in the app to add their own datasets:

- Change the original NIPA implementation to accept loading by file path or text csv content.
- Add a feature to add and manage datasets per user in the frontend app and orchestration server (a CRUD file upload would suffice). Store the files somewhere accessible for the ML service (for example, in cloud object storage like Amazon's s3, or decide on another strategy, like having the ML-service communicate with the orchestration server to ask for files). Also take into account that geographic data for visualization may also need to be uploaded

by the user per dataset to the orchestration server (which persists all these files in the database), but this only needs to be accessible to the frontend app.

- The MCP service can largely stay the same; maybe adding a tool to expose which datasets a user has available is helpful.

- The frontend needs to load the geographic visualization data per dataset, which can be done via a sort of global cache similar to the "useAuth" hook pattern, or by refactoring the code to use Tanstack's ReactQuery (which would effectively implement a similar pattern but may be easier to write or overall better). The geographic visualization data that should be used needs to be passed to the visualization components.

8.4.2 Lack of Personal Data Management

Users cannot currently request to delete their data (chats or accounts). This poses a possible privacy risk and could be addressed in the future. To address this limitation, the ML service layer needs to be modified to implement functionality to delete the NIPA operations; the frontend needs to be modified to allow this option in the UI; and the orchestration server needs to accept this from the frontend and delete the corresponding data in the database and the ML service (which currently uses local files for persistence and not the same database as the orchestration server).

8.4.3 Lack of an Administration Page

There is currently no administrator account or administration page. This makes it difficult to act as an administrator of the website without modifying databases outside of the app (manually). An administration page could benefit management of accounts in an organization.

8.4.4 Increased Security

Detecting suspicious login attempts and notifying users of these could benefit the application. However, this was not implemented. Additionally, email verification, password resetting, or two-factor authentication was not implemented, but it could benefit the application in the future.

8.4.5 User Experience Improvements

User experience improvements such as scrolling to the end of the chat, having chats be accessible by URL instead of being solely accessed by client interactivity, and disabling the new chat button when in an empty chat could make the app more aligned with modern user expectations, more intuitive to use, and improve the user experience.

Appendix A

Requirement Specification

A. Must-Have
1. As a user, I want to create an account using my email and password.
1.1 The system must allow users to create an account using only an email address and a password. 1.2 The system must require a password confirmation to ensure accuracy. 1.3 The system must hash the user's passwords before storing them in the database using a secure hashing algorithm. 1.4 The system must complete login and account creation operations within 2 seconds under normal load conditions.
2. As a user, I want to log in to the system.
2.1 The system must allow users to log in using their registered email and password. 2.2 The system must verify credentials and deny access if the email or password is incorrect. 2.3 The system must prevent brute-force attacks by limiting login attempts.

3. As a user, I want to input analytical goals in natural language.
3.1 The system must accept high-level analytical goals expressed in natural language. 3.2 The system must allow users to define analytical goals and constraints via the UI. 3.3 The agent must translate user goals into structured, multi-step NIPA workflows. 3.4 The system must provide a web-based chat interface for interacting with the agentic AI. 3.5 The agent must iteratively refine workflows based on results or user feedback.
4. As a user, I want to visualize the simulation results.
4.1 The agent must compare simulation strategies based on quantitative metrics such as peak infection value, total infected population, and mean squared prediction error. 4.2 The system must visualize the inferred infection network interactively. 4.3 The system must store chats.

Table A.1: Must-Have Classification

B. Should-Have
1. As a user, I want to see the AI agent's actions.
1.1 The system should allow users to inspect and reproduce previous simulation runs. 1.2 The system should log all agent actions, tool calls, and parameters used in simulations with timestamps for auditing and debugging. 1.3 The system should make agent decisions and actions inspectable by users. 1.4 The system should prevent unauthorized tool execution by the agent or external clients.
2. As a user, I want the dashboard to be easy to use without programming experience.
2.1 The system should be usable by non-expert users with limited epidemiological modeling experience. 2.2 The UI should prioritize clarity, interpretability, and transparency of agent behavior.
3. As a user, I want simulations to run efficiently.
3.1 The system should execute NIPA simulations within reasonable time bounds for interactive users. 3.2 The system should ensure that, for identical natural language inputs, it generates the same workflow and simulation outputs within a numerical tolerance of 1% for predicted infection probabilities.

Table A.2: Should-Have Classification

C. Could-Have
1. As a user, I want to log out of the system.
1.1 The system should allow users to log out.
2. As a user, I want meaningful error messages in case of wrong inputs.
2.1 The system should provide clear messages when the user tries to log/sign in. 2.2 The system should provide meaningful feedback and error messages to users. 2.3 The system should handle invalid or missing inputs, returning descriptive error messages without crashing. 2.4 The system should validate all inputs before executing NIPA-related operations.

Table A.3: Could-Have Classification

Appendix B

Gantt Chart

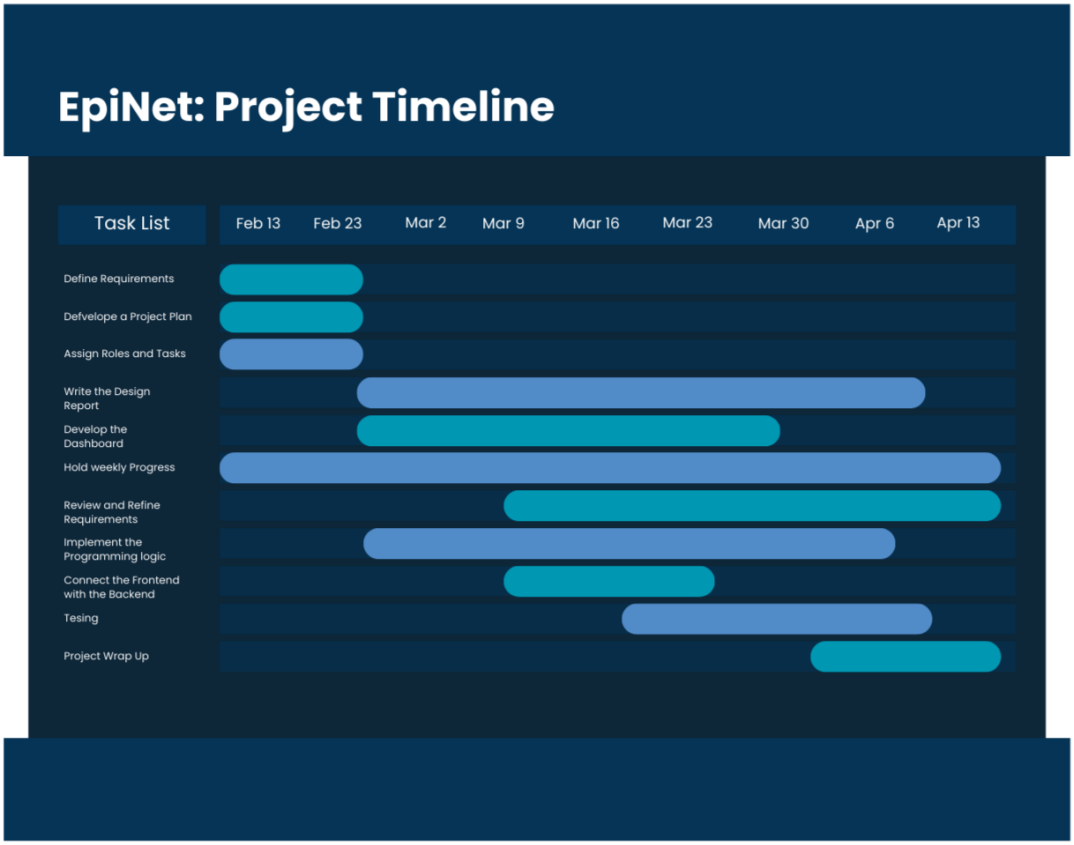

Figure B.1: Gantt Chart for the project timeline

Appendix C

Use Case Diagram

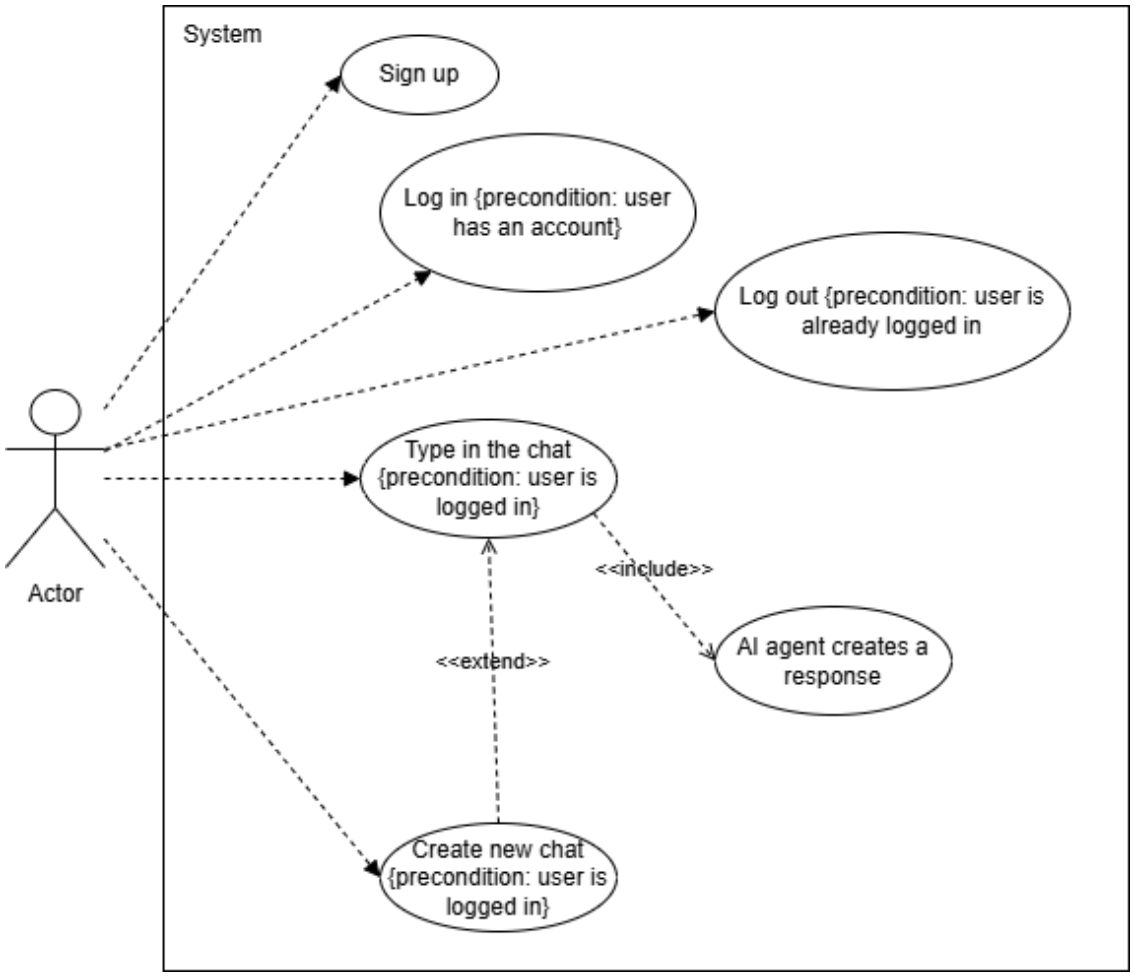

Figure C.1: Use Case Diagram

Appendix D

Activity Diagram

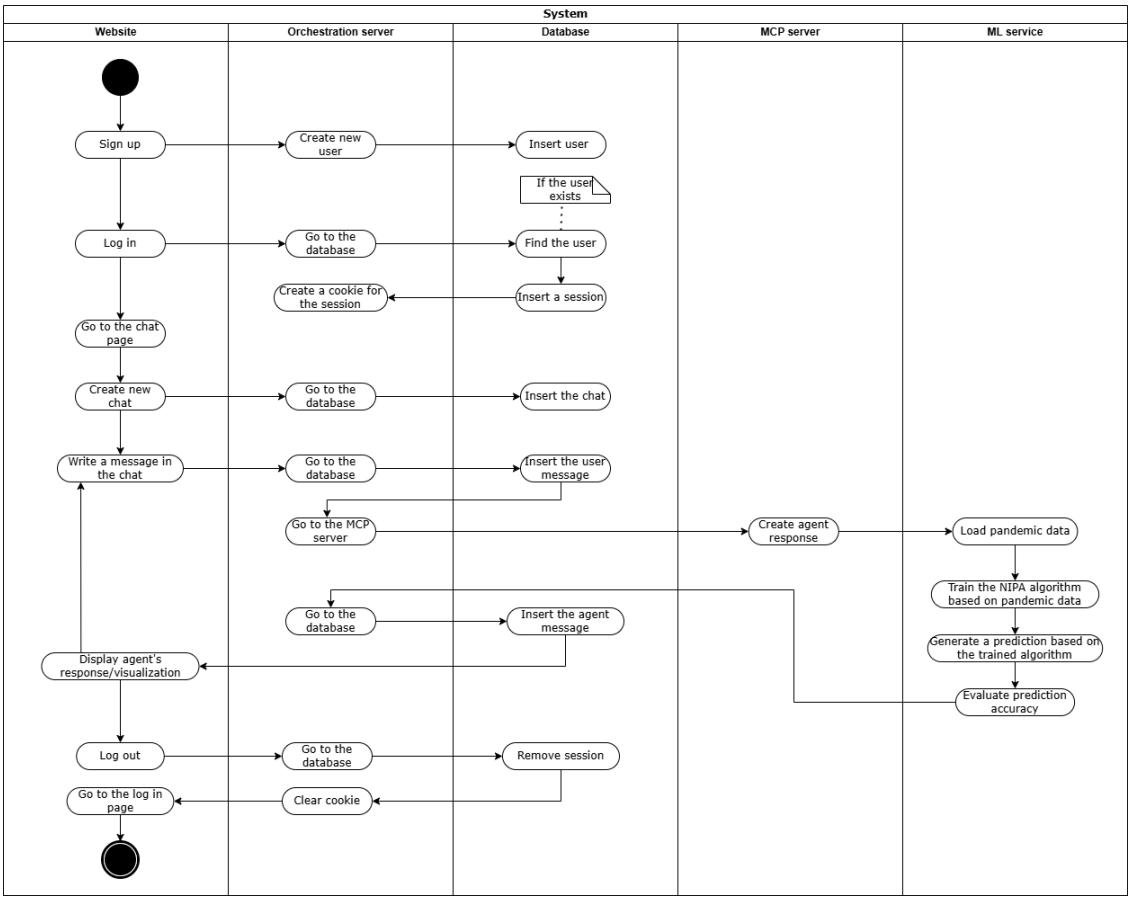

Figure D.1: Activity Diagram

Appendix E

AI Statement

During the preparation of this work, the authors Daria Hesson, Konstantinos Zygouris, Daniel Murse-Caraian, Gela Tsuladze, and Ashkan Nikjouyan used ChatGPT and QuillBot in order to rewrite original ideas more formally, give additional information to use in their own writing, and help with paraphrasing. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the work.

Bibliography

Ding, K., Yu, J., Huang, J., Yang, Y., Zhang, Q. & Chen, H. (2025), 'Scitoolagent: A knowledge graph-driven scientific agent for multi-tool integration'. https://arxiv.org/abs/2507.20280.

Dybå, T., Dingsøyr, T. & Moe, N. B. (2014), 'Agile project management'. https://doi.org/10.1007/978-3-642-55035-5_11.

Firesmith, D. (2003), 'Modern requirements specification', *Journal of Object Technology* **2**(2), 53–64. https://www.academia.edu/download/67562099/Modern_Requirements_Specification20210608-12436-v1fi3e.pdf.

Government of Mexico (2021), 'Covid-19 mexico data portal'. https://datos.covid-19.conacyt.mx/%7D.

Mannion, M. & Keepence, B. (1995), 'Smart requirements', *ACM SIGSOFT Software Engineering Notes* **20**(2), 42–47. https://dl.acm.org/doi/abs/10.1145/224155.224157.

Pham, T. D., Tanikanti, A. & Keçeli, M. (2025), 'Chemgraph: an agentic framework for computational chemistry workflows'. https://arxiv.org/abs/2506.06363.

PVS-Studio (2023), 'Software development lifecycle (sdlc)'. https://pvs-studio.com/en/blog/terms/6732/.

Ruparelia, N. B. (2010), 'Software development lifecycle models', *ACM SIGSOFT Software Engineering Notes* **35**(3), 8–13. https://doi.org/10.1145/1764810.1764814.

Srivastava, A., Bhardwaj, S. & Saraswat, S. (2017), 'Scrum model for agile methodology', pp. 864–869. https://doi.org/10.1109/CCAA.2017.8229928.

Sänger, M., De Mecquenem, N., Lewińska, K. E., Bountris, V., Lehmann, F., Leser, U. & Kosch, T. (2024), 'A qualitative assessment of using chatgpt as large language model for scientific workflow development', *GigaScience* **13**. https://doi.org/10.1093/gigascience/giae030.

Vijayakumar, S. (2024), 'Assessing the effectiveness of moscow prioritization in software development: A holistic analysis across methodologies', *EAI Endorsed Transactions on Internet of Things* **10**. https://doi.org/10.4108/eai.25-4-2024.204116.