Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY

Agent-Based Simulation and 3D Visualization of a Logistics Network

Deliverable 2

Mateusz Polis • Mateusz Nędzi

3.11.2025

Contents

1 Abstract

This document presents the requirements for our bachelor's thesis project of an *Agent-Based Simulation and 3D Visualization of a Logistics Network*. Functional requirements are specified in the form of use case descriptions and diagrams. Non-functional requirements covering Usability, Reliability, Performance and Supportability are enumerated in a table. A detailed schedule explains the steps we need to take to complete the project. Deliverables which we plan to deliver for each development and testing milestone are explained. Finally, we conducted an analysis of Strengths, Weaknesses, Opportunities and Threats, to identify potential future issues and be prepared to handle them.

1.1 History of changes

Date	Author	Description	Version
16.10.2025	M. Nędzi, M. Polis	Initial document revision	1.0
01.11.2025	M. Nędzi	Frontend Architecture and Technology Selection	2.0
01.11.2025	M. Nędzi, M. Polis	Communication	2.1
02.11.2025	M. Polis	Backend Architecture and Technology Selection,	2.2

2 Vocabulary

Map — A data structure representing the complete logistics network modeled as a directed multigraph.

Edge — A connection between two Nodes in the Map's graph, representing a traversable route (e.g., road). Edges are directed, defining the allowed direction of movement between Nodes. Each edge includes attributes relevant to the simulation of traffic flow—such as distance, capacity and maximum speed—along with any additional parameters necessary to capture route-specific conditions. Edges may also be constrained (e.g., weight/height limits). Agents move along the edges. Events like delays or blockages are modelled on edges.

Node — A vertex in the Map's graph. Nodes mark where Edges meet. A node can correspond to one or more physical locations—Buildings—such as warehouses, depots etc. Each node carries geographic coordinates.

Agent — An autonomous entity within the simulation that perceives its environment and acts according to defined behavioral rules. Agents may represent mobile entities (moving along the Edges), stationary entities (Buildings located at Nodes), or external entities not directly represented on the Map (e.g., a Broker agent coordinating routes). Each agent type possesses its own set of attributes and decision-making logic, allowing for dynamic interactions and emergent behaviors within the simulated Logistics Network.

Event — An occurrence within the simulation that represents a change of state in the Logistics Network. Events may take place at Nodes, within Buildings, along Edges, or for specific Agents. They capture dynamic phenomena such as vehicle arrivals, loading and unloading operations, traffic delays, equipment failures or decision triggers. Each event is characterized by a type, time, and affected entities. Events can occur deterministically or with a degree of randomness to reflect the uncertainty and variability of real-world logistic processes, enhancing the realism of the simulation.

Building — A physical facility located at a Node within the Map. Buildings represent logistic infrastructure such as warehouses, depots or retail outlets. A Building may function as an Agent,

actively participating in the simulation (e.g., managing inventory, dispatching vehicles), but it does not have to. Some may serve as passive locations or resources. Each building possesses its own set of parameters relevant to its role, such as storage capacity, processing rate, operational hours, or handling costs. However, these parameters are not common across all Buildings, as each type exhibits distinct behavior and functionality within the simulated Logistics Network.

Logistics Network — The environment in which all simulation activity takes place, representing the interconnected system of transportation routes, facilities and operational entities responsible for the movement of goods. The Logistics Network is composed of Nodes, Edges, and Buildings, forming the structural backbone of the simulation's Map. Within this environment, the Agentic System operates. The Logistics Network serves both as a physical model of infrastructure and as a context in which agent behaviors and system-wide logistics processes emerge.

Agentic System — A multi-agent environment operating within the Logistics Network, composed of autonomous Agents that perceive their surroundings, make decisions and act according to predefined or adaptive behavioral rules. The Agentic System governs the interactions between Agents such as Vehicles, Buildings or Brokers and their responses to Events occurring across the Map. Through these interactions, the Agentic System produces emergent behaviors that reflect the complexity of real-world logistics processes, such as congestion, resource allocation and coordination.

Frontend — A web-based interface responsible for visualizing and interacting with the simulation. It renders the 3D environment using Three.js [1] and provides an interactive overlay built with React [2] for controls, information panels, and event details. The Frontend communicates with the Backend through a WebSocket [3] connection, sending *Actions* that can influence the simulation and receiving *Signals* that describe state changes and events to be displayed.

Backend — A server-side system responsible for executing the Agentic Simulation and managing the overall logic of the Logistics Network. The Backend maintains the state of all Agents, processes Events, and produces Actions that describe updates to be reflected in the Frontend. It communicates with the Frontend through a WebSocket connection, continuously sending Signals and receiving Actions that represent user commands or parameter changes. The Backend ensures deterministic simulation flow, consistency across connected clients, and real-time synchronization between the computational model and its 3D visualization.

Action — A message sent from the Frontend to the Backend through the WebSocket connection to influence the simulation. Actions represent user commands or parameter changes, such as modifying an Agent's behavior, removing an Element, or controlling the simulation state (e.g., play, pause, reset).

Signal — A message sent from the Backend to the Frontend describing state changes or events within the simulation. Signals inform the Frontend what has occurred—such as an Agent moving, an Event being triggered, or metrics being updated—so that it can update the 3D visualization accordingly.

Element — An interactive object represented on the Map that can be selected by the user to inspect its details. Elements include all simulation-relevant components such as Buildings, Vehicles, and other Agents that influence or participate in the Logistics Network. Each Element contains information describing its current state, parameters and recent Events. Non-functional visual objects such as trees, terrain or rivers are not considered Elements, as they serve only a decorative or contextual purpose and do not interact with the simulation logic.

Fleet — A collection of mobile Agents within the simulation, typically representing vehicles, such as trucks, that move along the Edges of the Map. The Fleet forms the dynamic component of the Logistics Network responsible for executing transport and delivery operations.

3 Specification

3.1 Executive Summary

Our project involves a web application for a truck logistics company that simulates traffic across a logistics network with respect to truck CO_2 emissions, capacity, speed and fuel consumption. The frontend emphasizes accessibility and ease of use, featuring in-browser 3D visualization, while the backend employs an agent-based simulation to ensure accuracy. To support flexible experimentation, the system includes a map editor for building custom networks and tools to define fleets of trucks with user-specified properties for running simulations.

3.2 Functional Requirements

We specify functional requirements in a form of use case descriptions and a use-case diagram.

3.3 Functional Requirements

We specify functional requirements in a form of use case descriptions and a use-case diagram.

Table 2: Description of use cases of interaction of the user with the system

ID	Actor	Name	Description	System Response
UC1	User	Define Logistics Network	The user specifies the structure of the logistics system, including nodes, edges and buildings.	The system creates a new logistics network structure and visualizes it on the map.
UC2		Define Map	The user defines the map topology, adding nodes and edges with parameters such as distance, capacity, and speed limits.	The system registers the map structure and displays it in the 3D environment.
UC3		Define Fleet	The user defines the fleet composition.	The system adds the defined vehicles as agents to the simulation environment.
UC4		Define Truck	The user configures an individual truck kind to be used in the fleet, setting its parameters such as speed, capacity, and starting location.	The system creates the truck as an agent and associates it with the logistics network.
UC5		Interact With Simulation	The user selects and inspects elements or events directly from the 3D visualization.	The system displays an overlay panel with element details, attributes, and/or recent events.
UC6		Control Simulation Playback	The user controls the simulation state using playback buttons (play, pause, stop).	The frontend sends a corresponding action to the backend, and the simulation updates accordingly.
UC7		Speed Up Simulation	The user increases the simulation playback speed.	The system adjusts the simulation timestep and updates the visualization rate.
UC8		Slow Down Simulation	The user decreases the simulation playback speed.	The system adjusts the simulation timestep to a slower rate and updates accordingly.

ID	Actor	Name	Description	System Response
UC9		Start Simulation	The user initiates the simulation run.	The backend starts the simulation engine, initializes all agents, and begins sending signals to the frontend.
UC10		Pause Simulation	The user pauses an ongoing simulation.	The backend halts simulation progression while preserving current agent states.
UC11		Lookup Simulation State	The user requests an overview of the current global simulation state (e.g., time, active agents, events).	The system retrieves and displays the current simulation metrics and timeline state.
UC12		Lookup Element State	The user requests detailed information about a specific agent or element.	The system returns detailed attributes, parameters, and status of the selected entity.
UC13		Control Camera	The user interacts with the 3D view using mouse or keyboard to rotate, pan, and zoom.	The system updates the camera position and orientation in real time.
UC14		Control Camera Position	The user moves the camera to a predefined or selected element position.	The system transitions the camera smoothly to focus on the selected location or element.
UC15		Control Camera	The user adjusts the viewing angle or perspective within the 3D environment.	The system applies the new camera parameters and updates the visualization.
UC16		Export Logistics Network	The user exports the defined logistics network configuration.	The system saves the current network (nodes, edges, buildings) to a file.
UC17		Import Logistics Network	The user imports a saved logistics network file.	The system parses the data, recreates the map, and visualizes it within the scene.
UC18		Export Simulation State	The user exports the current simulation state, including agent positions and parameters.	The system serializes the simulation data and saves it to an external file.
UC19		Import Simulation State	The user loads a previously saved simulation state.	The system restores agent states, simulation time, and environment conditions to resume execution.

3.4 Non-functional requirements

The list of non-functional requirements in shown in the table 4.

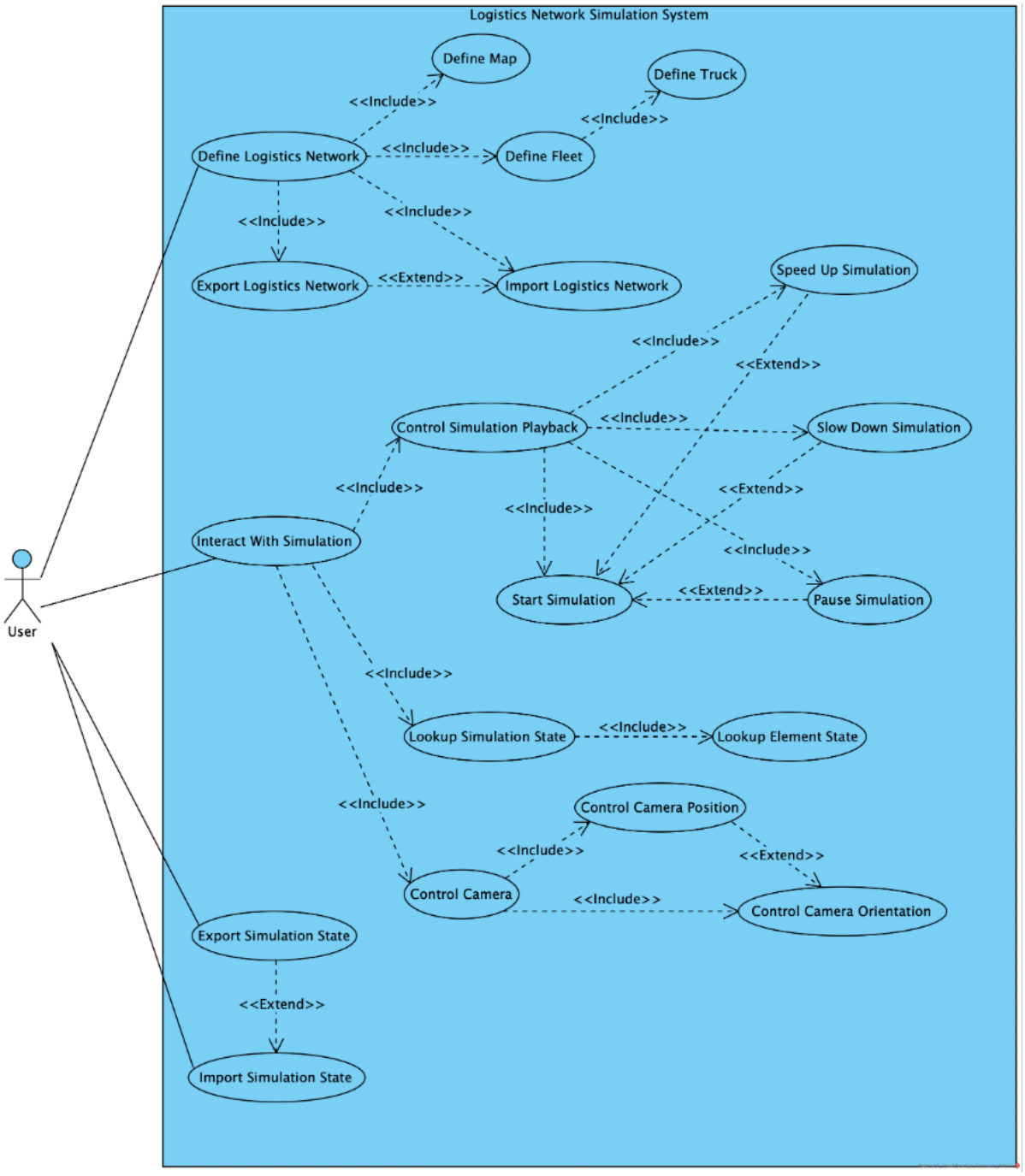

Figure 1: Use-case diagram showing interaction of the user with the system.

Requirements area	**No.**	**Description**
Utility (Usability)	1	Functional and responsive interface. The Frontend adapts dynamically to various screen sizes and resolutions while maintaining full functionality and readability. It must operate correctly on displays with a minimum resolution of 1280×720 (HD).
Utility (Usability)	2	Intuitive navigation. The user interface must be self-explanatory and easy to learn without formal training. Users should be able to perform basic interactions such as selecting Elements, viewing details, or controlling simulation without external documentation.
Reliability	3	Crash recovery. The Backend should handle unexpected client disconnections gracefully, preserving simulation continuity for other connected users.
Reliability	4	Network failure resistance. The system must remain stable in the event of temporary network disruptions. The Frontend automatically attempts to reconnect to the Backend through the WebSocket connection and restores synchronization without user intervention.
Reliability	5	Uptime requirement. The system should maintain an operational availability of at least 99% during planned usage periods.
Reliability	6	Error reporting. All errors and connection issues should be logged and reported in a structured way for debugging and diagnostics.
Performance	7	Simulation capacity. The application must support simulations of logistics networks containing up to 100 Nodes and 20 active Agents while maintaining stable performance and real-time responsiveness.
Performance	8	Rendering efficiency. The Frontend should maintain a minimum frame rate of 30 FPS under typical simulation conditions and avoid noticeable lag when visualizing agent movement or events.
Maintenance (Supportability)	9	Testing coverage. Core simulation logic and communication protocols should include automated unit tests to ensure consistent behavior across updates.
Maintenance (Supportability)	10	Clear codebase structure. The software codebase must be organized in a logical and consistent directory structure, with clear separation between modules. File organization should facilitate readability, maintainability, and team collaboration.
Maintenance (Supportability)	11	Environment independence. The entire system should be deployable in a Docker container environment to ensure consistent behavior across different operating systems and hardware configurations. Containerization simplifies setup, testing, and maintenance by encapsulating all dependencies and runtime environments.

4 Project schedule

The project is planned to be implemented per the following schedule shown on the figure 2.

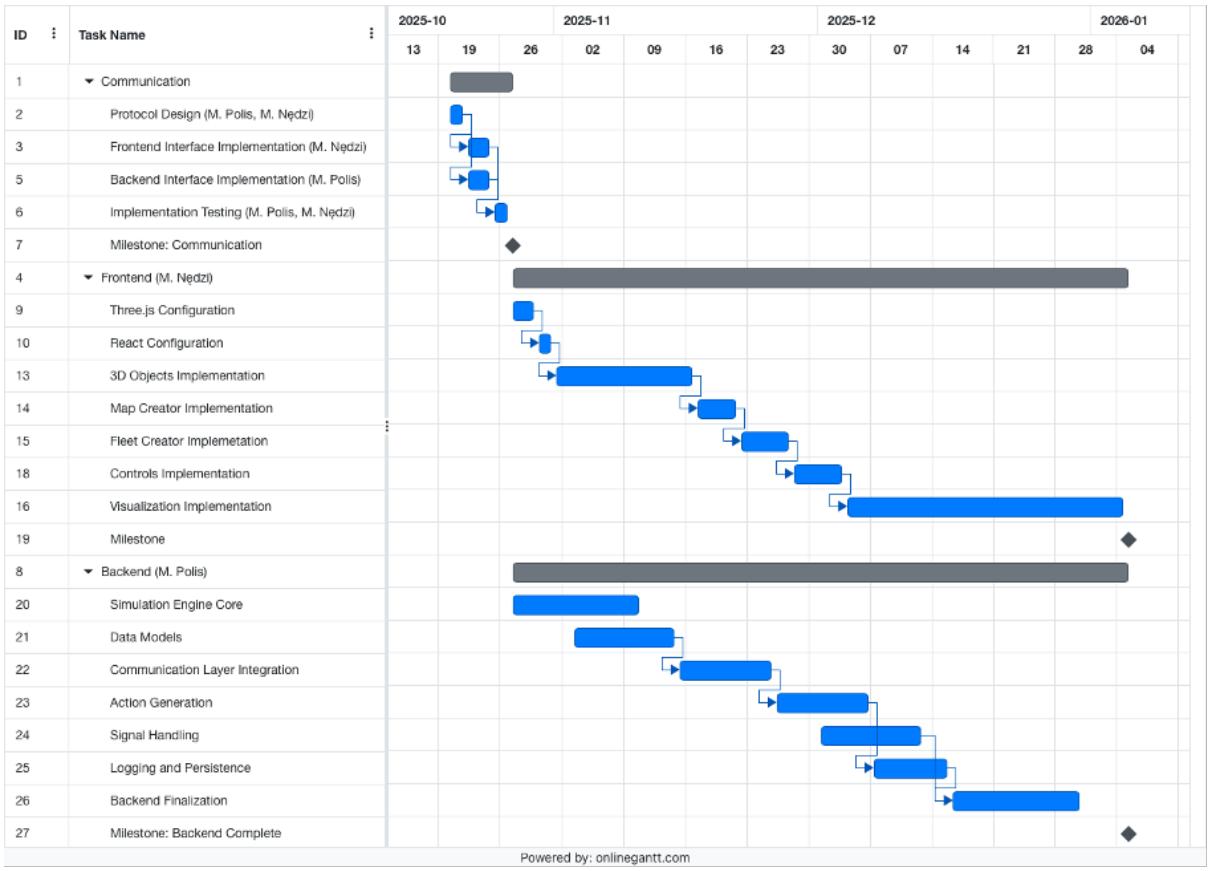

Figure 2: Project schedule, Gantt diagram.

5 Risk Analysis

SWOT Overview

	Threats	Opportunities
Internal	• Limited time due to other academic projects, jobs, thesis-related work. • Lack of prior experience with 3D visualization frameworks (Three.js) and WebSocket-based communication. • Integration issues between simulation logic (Backend) and visualization (Frontend).	• Opportunity to gain hands-on experience in full-stack system design combining simulation and web technologies. • Possibility to reuse developed components for future research or educational purposes. • Strengthening collaboration and project management skills within a software team.

External	• Unexpected compatibility issues between browsers, operating systems, or hardware when running 3D rendering. • Potential delays due to dependencies on external libraries or APIs.	• Technological advancement and open-source support for agent-based modeling and 3D visualization frameworks. • Scalability for further development into an interactive decision-support tool.

Threats: Likelihood, Impact, Mitigation

Threat 1 — Limited time due to other academic commitments and jobs.
Likelihood: High *Impact:* Medium
Mitigation: Create a detailed schedule with milestones and assign responsibilities early. Hold short weekly progress reviews to maintain steady development pace.

Threat 2 — Lack of experience with 3D visualization and WebSocket communication.
Likelihood: Medium *Impact:* Medium
Mitigation: Begin with a small prototype integrating Three.js and WebSocket early in the project. Allocate time for learning key libraries and reviewing example implementations.

Threat 3 — Integration challenges between Backend and Frontend.
Likelihood: Medium *Impact:* High
Mitigation: Define a clear data exchange protocol (Actions/Signals) early and test integration continuously rather than at the final stage.

Threat 4 — Browser or hardware compatibility issues.
Likelihood: Low to Medium *Impact:* Medium
Mitigation: Test the application across multiple browsers (Chrome, Firefox, Safari) and devices; use fallback rendering modes and responsive design principles.

Threat 5 — External dependency delays (library updates or bugs).
Likelihood: Low *Impact:* Medium
Mitigation: Freeze library versions during development and document dependency versions for reproducibility.

6 System Architecture

6.1 Frontend

Frontend employs a layered architecture that cleanly separates concerns between network communication, domain logic, and rendering. The system implements an **IO → Domain → View** pattern that maintains strict boundaries between each layer. This separation ensures that simulation state remains pure and serializable while rendering logic remains isolated from business rules, enabling independent testing, maintenance, and evolution of each layer.

6.1.1 Layered Architecture

The architecture is organized into four primary layers, each with distinct responsibilities:

Input/Output Layer This layer handles all communication with the backend over WebSocket. It validates incoming messages using runtime schema validation, decodes protocol envelopes, and emits strongly-typed events to the rest of the system. Beyond message

handling, this layer manages connection lifecycle, implements exponential backoff strategies for reconnection, and handles request-response matching for asynchronous operations. By isolating all network concerns, the system can adapt to different transport mechanisms or protocols without affecting domain logic.

Domain Layer This layer maintains pure, serializable simulation state with no dependencies on rendering or UI frameworks. The domain represents simulation entities and their relationships using immutable data structures. Updates arrive as ordered events (typically a sequence of entity updates followed by a tick marker), which are accumulated into a working state. When a tick completes, the working state is frozen into an immutable snapshot and stored in a temporal buffer. This design enables time-travel debugging, replay capabilities, and deterministic state management. An interpolation system generates smooth frames between discrete snapshots, allowing the rendering layer to display fluid motion even when network updates are sparse.

Rendering Engine This layer provides the infrastructure for three-dimensional visualization using WebGL. It manages the rendering loop, scene graph, camera, lighting, and material systems. The engine maintains a consistent visual style characterized by simplified geometry, flat shading, and warm lighting—creating a cohesive aesthetic that prioritizes clarity over photorealism. By abstracting rendering concerns from domain logic, the system can potentially swap rendering backends or adapt to different platforms without modifying simulation code.

View Layer This layer bridges the gap between immutable domain snapshots and the mutable scene graph required for rendering. It transforms domain entities into visual objects, updates their positions and properties based on interpolated frames, and manages the lifecycle of three-dimensional objects. Critically, this is the only layer permitted to mutate rendering objects, enforcing the architectural principle that domain state remains pure and reproducible.

An additional **User Interface layer** provides overlay controls and information displays built with modern web technologies. This layer operates independently from the rendering engine, allowing UI components to be developed, tested, and modified without affecting the core visualization pipeline.

6.1.2 Data Flow

The system processes simulation updates through a unidirectional data flow that decouples network timing from rendering performance:

1. **Message Reception**: Raw messages arrive from SPINE over WebSocket and are validated at the network boundary using schema validation. Invalid messages are rejected before entering the domain layer, protecting the system from malformed data.
2. **Protocol Translation**: Validated protocol messages are translated from network wire formats into domain events. This translation layer ensures that changes to the protocol structure do not propagate through the entire system, maintaining a stable domain model.
3. **State Accumulation**: Domain events are processed sequentially, accumulating updates into a working state. Entity updates (e.g., position changes, property modifications) are applied immediately, while tick markers indicate completion of a simulation step.
4. **Snapshot Creation**: When a tick completes, the accumulated working state is frozen into an immutable snapshot. This snapshot represents a complete, deterministic view of the simulation at that point in time. The snapshot is stored in a temporal buffer that maintains a history of recent simulation states.

5. **Temporal Interpolation**: The rendering loop operates independently from network updates, querying the snapshot buffer for states that bracket the current render time. The system computes an interpolation factor $\alpha \in [0, 1]$ representing where the current time falls between two snapshots. This allows smooth rendering even when network updates arrive irregularly or with latency.

6. **Visual Adaptation**: The interpolated state is transformed into visual objects suitable for rendering. Domain entities are mapped to their visual representations, positions are interpolated between snapshot states, and object properties are updated accordingly. This transformation preserves the immutability of domain snapshots while enabling smooth visual updates.

7. **Frame Rendering**: The rendering engine draws the scene at a consistent frame rate (typically 60 FPS), independent of simulation update frequency. This separation ensures smooth visual presentation regardless of network conditions or server tick rate.

This pipeline design provides several key benefits: network latency and jitter are absorbed by the snapshot buffer, rendering quality remains consistent, and the system can support features like pause, replay, and time-travel debugging by accessing historical snapshots. Data flow is also shown on the figure 3.

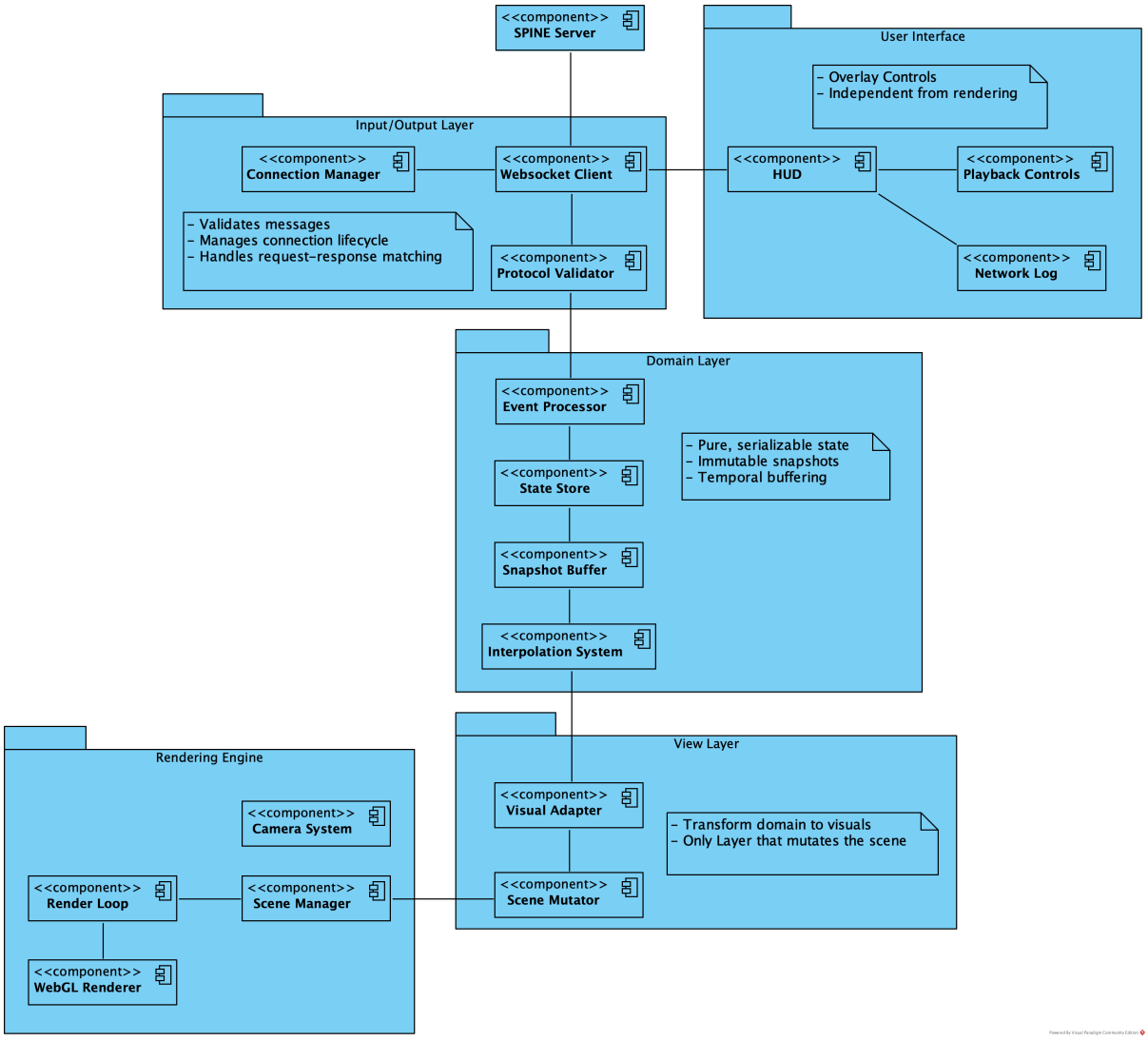

Figure 3: Components Diagram presenting data from and architecture of the frontend module

6.1.3 Architectural Constraints

The architecture enforces several key constraints that maintain separation of concerns and enable the system's key benefits:

- **Domain Purity**: The domain layer contains no rendering types, user interface APIs, or side effects. All domain types are serializable and can be persisted or transmitted over the network. This purity enables the domain to be tested independently, reused across different visualization contexts, and potentially executed in different environments (e.g., server-side validation or headless simulation).
- **Adaptation Boundaries**: Type conversions occur explicitly at layer boundaries. Network wire formats (JSON structures) are distinct from domain types (immutable simulation state) and view models (mutable rendering objects). This separation means that protocol changes do not propagate through the system, rendering changes do not affect domain logic, and domain evolution does not require rewriting visualization code.
- **Mutation Isolation**: Only the view layer mutates rendering objects. The domain layer produces immutable snapshots that can be safely shared, cached, and replayed. The rendering engine manages the scene graph and rendering resources but does not interpret simulation state. This isolation prevents accidental coupling between rendering performance optimizations and domain correctness.
- **Compositional Structure**: The system is composed from independent, loosely-coupled components. The entry point wires together these components but does not contain business logic or rendering code. This composition enables different configurations for different use cases (e.g., development vs. production, different visual styles, or alternative rendering backends) without modifying core components.

6.1.4 Key Design Patterns

Several design patterns are central to the architecture's effectiveness:

Event-Driven State Management The domain layer uses an event-driven approach where simulation updates arrive as discrete events rather than complete state transfers. Events are accumulated into a working state, and only when a tick completes is the state frozen into an immutable snapshot. This pattern enables efficient incremental updates, supports undo/redo operations, and facilitates debugging by maintaining a complete event log.

Temporal Buffering The snapshot buffer maintains a sliding window of recent simulation states, enabling the system to interpolate between discrete snapshots for smooth rendering. This buffer serves multiple purposes: it absorbs network jitter, enables replay functionality, supports seeking through simulation history, and allows the rendering loop to operate independently from network timing. The buffer capacity balances memory usage against the temporal history needed for smooth playback.

Immutable Snapshots Each simulation tick produces an immutable snapshot that represents a complete, deterministic view of the simulation state at that moment. These snapshots can be safely shared across threads, cached indefinitely, and replayed deterministically. This immutability is fundamental to enabling features like time-travel debugging, deterministic replay, and state comparison across different simulation runs.

Adapter Pattern The boundaries between layers use adapter functions that translate between different type systems. Network protocols are adapted to domain events, domain entities

are adapted to visual objects, and UI actions are adapted to domain commands. This pattern ensures that changes in one layer do not propagate to others, maintaining loose coupling while enabling each layer to evolve independently.

Request-Response Matching The network layer implements asynchronous request-response matching that handles out-of-order message delivery. This allows the system to send multiple requests concurrently and match responses correctly even when network timing varies. This pattern is essential for responsive user interactions in a distributed system where network latency is unpredictable.

6.1.5 Type Safety and Validation

The architecture employs a dual type system that provides both compile-time type checking and runtime validation. Network messages are validated against schemas at the system boundary, ensuring that invalid or malformed data cannot corrupt the domain state. This validation occurs before any domain processing, protecting the system from protocol errors, version mismatches, or malicious input. The type system ensures that developers catch errors early in development while maintaining runtime guarantees in production.

6.2 Backend

Backend provides the authoritative simulation engine, action processing pipeline, and real-time signaling over WebSocket. It is structured as a set of clearly separated layers that isolate network concerns, command validation, simulation control, and the world model. The design emphasizes deterministic ticks, explicit actions and signals, and robust boundary validation to ensure safety and observability.

6.2.1 Layered Architecture

The Backend is organized into cohesive layers with single responsibilities:

Interface Layer (WebSocket API) A FastAPI-based WebSocket service (served by Uvicorn) manages client connections, message reception, and broadcast. Incoming JSON messages are parsed with `orjson`; connections are tracked by a `ConnectionManager`. A lightweight HTTP `/health` endpoint supports liveness checks. This layer never mutates the simulation; it only parses, validates, and forwards requests or broadcasts signals.

Command Layer (Actions) Client requests use a stable `"domain.action"` string format with parameters. An `ActionParser` validates raw messages into typed `ActionRequest`s. An `ActionRegistry` maps actions to handler functions, implementing a Command pattern. Handlers are grouped by domain: simulation control, tick-rate, agent lifecycle, map import/export/generation, and state snapshot. All handlers execute against a minimal *HandlerContext* (simulation state, world, signal queue, logger).

Simulation Core A `SimulationController` owns the main loop and a thread-safe `SimulationState` (running/paused, tick rate, current tick). It drains the action queue, applies effects (e.g., start/pause, map operations, agent changes), and if running, advances the world one deterministic step per tick. Each step emits semantic signals that are consumed by the Interface Layer for broadcast.

World Model The `World` encapsulates agents, packages, and a directed road `Graph` (nodes/edges with classes, speeds, lanes, and optional weight limits). A single `step()` performs: perceive, message delivery, site processing (spawn/expiry), decide/act, and diff serialization for UI. The world also exposes full-state snapshots and safe import/export of maps.

I/O and Persistence The `map_manager` provides safe GraphML import/export into the repository `maps/` directory with name sanitization to prevent path traversal. Procedural map creation is provided by a hierarchical `MapGenerator` with validated `GenerationParams`.

Runtime Orchestration A `SimulationRunner` wires the world, controller, queues, and WebSocket server. It starts the controller in a background thread and the WebSocket server in another thread with its own asyncio loop, handles shutdown signals, and exposes a small status probe for operational visibility.

6.2.2 Data Flow

The Backend processes requests and produces signals through a unidirectional, decoupled pipeline:

1. **Client Action** arrives over WebSocket as JSON with fields `action` and `params`. The server parses with `orjson` and validates with `ActionParser`, enforcing the `"domain.action"` format.
2. **Enqueue**: Validated `ActionRequest`s are enqueued into a thread-safe `ActionQueue`. Invalid JSON or schema errors are immediately answered with an error message to the originating connection.
3. **Dispatch**: The `SimulationController` drains the `ActionQueue`. For each action, `ActionRegistry` locates a handler and executes it with a `HandlerContext`. Handlers may mutate `SimulationState` and `World`, and emit domain signals.
4. **Tick**: If `state.running` and not paused, the controller increments the tick, emits `tick.start`, calls `World.step()`, processes step results (world events and agent diffs), and emits `tick.end`.
5. **Broadcast**: Signals are put into a thread-safe `SignalQueue`. An async task dequeues signals, serializes them (ensuring string keys), and broadcasts to all active WebSocket clients.
6. **State Snapshots**: On `simulation.start`, `state.request`, or new client connections while running, a bounded sequence of snapshot signals is sent: `state.snapshot_start` → `state.full_map_data` → multiple `state.full_agent_data` → `state.snapshot_end`.

This design decouples network I/O timing from simulation timing, protects the core loop from slow clients, and yields deterministic tick progression.

6.2.3 Concurrency and Timing

- **Threads**: The controller runs in a dedicated thread; the WebSocket server runs in another thread with its own asyncio event loop; queues bridge the two.
- **Tick Rate**: The `SimulationState` clamps tick rate (e.g., 0.1–100 Hz) and the controller sleeps based on the configured rate when running.
- **Thread Safety**: `SimulationState` uses a lock for all reads/writes; queues are thread-safe; world mutation occurs only on the controller thread.
- **Graceful Shutdown**: OS signals trigger orderly stop of controller, server, and async broadcast task.

6.2.4 Architectural Constraints

- **Command/Signal Contract**: All requests follow `"domain.action"`; all outbound messages follow `"domain.signal"` with a uniform `{ signal, data }` envelope.

- **Boundary Validation**: Actions are validated before entering the core. Map operations sanitize filenames and require the simulation to be stopped; generation parameters are fully checked before use.
- **Deterministic Ticks**: World updates happen only on the controller thread between `tick.start` and `tick.end`.
- **Isolation**: The Interface Layer never mutates simulation state; it only enqueues actions and broadcasts signals.

6.2.5 Key Design Patterns

Command Registry Actions are mapped to handlers via `ActionRegistry`, enabling modular extension: new capabilities are added by registering a new `"domain.action"` and its handler.

Event-Driven Signaling The controller and handlers emit semantic signals that the WebSocket layer serializes and broadcasts. Clients consume a consistent stream of `tick.*`, `simulation.*`, `map.*`, `state.*`, `agent.*`, and package/site-related signals.

Producer–Consumer Queues Thread-safe queues decouple producers (WebSocket input, controller) from consumers (controller, broadcaster), preventing backpressure from stalling the loop.

Snapshot on Connect New clients can receive a bounded snapshot sequence to reconstruct full state without halting the simulation or requiring ad hoc API calls.

6.2.6 Type Safety and Validation

The Backend uses Pydantic models and enums to provide runtime validation and consistent serialization:

- **ActionRequest** validates the `"domain.action"` format and enforces `params` to be a dictionary.
- **Action/Signal Models** provide typed structures with field validation; signals are serialized as `{ signal, data }` using an overridden `model_dump` for consistency.
- **GenerationParams** validates all map generation parameters, including ranges and relationships (e.g., [] ordering, probability bounds).
- **Name Sanitization** prevents path traversal in map export/import. JSON serialization ensures string keys at all nesting levels for `orjson`.

6.2.7 Public API (Selected)

Requests are sent as actions; responses are broadcast as signals. Representative examples:

- **simulation.start** → `simulation.started` plus a snapshot sequence
- **simulation.pause/resume/stop** → `simulation.paused/resumed/stopped`
- **tick_rate.update** → running tick rate update, no pause required
- **agent.create/delete/update** → `agent.updated` diffs during ticks
- **map.import/export/create** → `map.imported/exported/created` (only while stopped)
- **state.request** → snapshot sequence `state.snapshot_start`, `state.full_`, `state.snapshot_end`

6.2.8 Error Handling and Reliability

Invalid actions or parameters yield an `error` signal with message and optional tick. The controller isolates unexpected exceptions per-iteration and continues after brief backoff. On network errors, failed connections are pruned; shutdown cancels the broadcaster, joins threads, and sets `uvicorn.should_exit`.

6.2.9 Operations

A CLI entrypoint starts the runner with host/port/log-level flags. Logging is centralized and consistent across components. A simple `/health` GET endpoint supports readiness checks during deployment.

6.2.10 Extensibility

New capabilities are added by registering a `"domain.action"` with a handler and, if needed, introducing new `"domain.signal"` emissions. Map generation and world logic are modular; additional handlers (e.g., routing, traffic dynamics) can be added without touching the Interface Layer or controller scaffolding.

7 Communication

The communication between the Frontend (**VISTA**) and Backend (**SPINE**) is implemented using a **WebSocket** protocol. This architecture was chosen to provide a bidirectional, low-latency communication channel suitable for real-time simulation synchronization and visualization updates. Unlike traditional REST APIs, which rely on discrete HTTP requests, the WebSocket connection allows both components to continuously exchange messages without re-establishing a connection for each operation.

7.1 Rationale for Using WebSocket

The simulation environment requires a constant stream of updates to reflect agent movements, state changes, and triggered events within the 3D visualization. A polling-based approach using REST would be inefficient and introduce unnecessary latency. WebSocket ensures that:

- **Real-time communication** is maintained between client and server, ensuring that all simulation state changes are propagated immediately.
- **Low overhead** is achieved through persistent connection, avoiding repetitive HTTP handshakes.
- **Bidirectional messaging** allows both systems to send and receive updates dynamically, enabling user-triggered actions (e.g., play, pause, map import) and system-driven updates (e.g., simulation tick updates, agent movement).
- **Consistency** is maintained across clients since all state updates are distributed directly through a single synchronized channel.

This approach is essential for maintaining smooth real-time feedback and a responsive 3D interface, especially as the simulation grows in complexity or involves multiple simultaneous users.

7.2 API Protocol

The WebSocket protocol defines a structured message exchange pattern between VISTA and SPINE. Each message belongs to one of two primary types: **Action** or **Signal**.

Actions

Actions are messages initiated by the Frontend (VISTA) and sent to the Backend (SPINE) to request a specific operation or change in simulation state. Each Action has a unique identifier in the form `<domain>.<action>` and includes an optional set of parameters.

```
{
  "action": "<domain>.<action>",
  "params": {
    "param_1": "param_1_value",
    "param_2": "param_2_value"
  }
}
```

Examples of actions include:

- `simulation.start`, `simulation.stop`, `simulation.resume`
- `map.create`, `map.import`, `map.export`
- `agent.create`, `agent.update`, `agent.delete`, `agent.get`
- `tick_rate.update`

Signals

Signals are messages initiated by the Backend (SPINE) and sent to the Frontend (VISTA) to report system state changes, results of actions, or internal events. They follow the same naming convention, `<domain>.<signal>`, and include an optional `data` field.

```
{
  "signal": "<domain>.<signal>",
  "data": {
    "param_1": string,
    "param_2": number
  }
}
```

Signals are used to confirm successful operations, update UI elements, or propagate simulation events. Examples include:

- `simulation.started`, `simulation.stopped`, `simulation.resumed`
- `map.created`, `map.imported`, `map.exported`
- `agent.created`, `agent.updated`, `agent.deleted`, `agent.state`
- `event.created`, `building.updated`
- `error`

Example Message Exchange

Starting a Simulation When the user initiates a simulation run from the Frontend:

```
# Action sent by VISTA:
{
  "action": "simulation.start",
  "params": {
    "tick_rate": 60
  }
}

# Signal from SPINE:
```

```
{
  "signal": "simulation.started",
  "data": {
    "tick_rate": 60
  }
}
```

Creating a Map

```
# Action:
{
  "action": "map.create",
  "params": {
    "size": 100
  }
}

# Signal:
{
  "signal": "map.created",
  "data": {
    "size": 100
  }
}
```

Handling Errors If any operation fails, SPINE sends a standardized error signal:

```
{
  "signal": "error",
  "data": {
    "code": "INVALID_REQUEST",
    "message": "Unknown action parameter"
  }
}
```

Design Benefits

- The protocol enforces a consistent, domain-scoped naming convention across all messages.
- JSON-based serialization ensures human readability and ease of integration with external clients.
- All actions are confirmed by corresponding signals, which guarantees reliable synchronization between VISTA and SPINE.
- The uniform structure allows extension for new domains or simulation features without breaking compatibility.

Overall, the WebSocket-based API protocol provides a robust and extensible communication layer for real-time synchronization between the simulation engine and its 3D visualization interface.

8 User's Activity

User is supposed to define a map and a fleet to start a simulation. Next, they can perform various interaction with the running simulation as long as they want to as well as pausing the

simulation. Show on the figure 4.

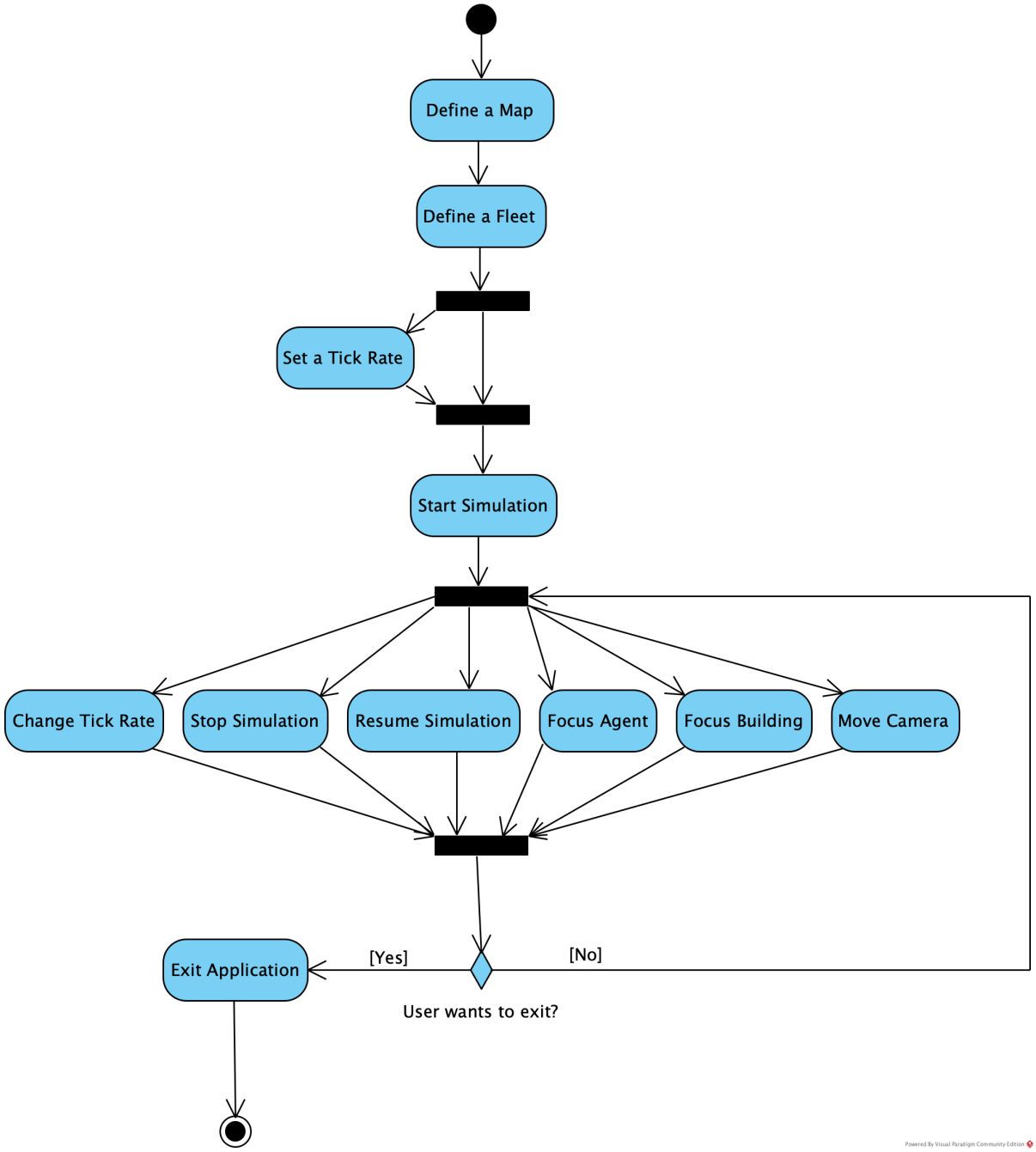

Figure 4: Activity diagram presenting user's interaction with the app

9 Simulation State Class Diagram

The simulation state represents the complete and evolving snapshot of the logistics network at any given simulation tick. It encapsulates both the static topology of the system—such as nodes, roads, and buildings—and its dynamic elements, including agents, vehicles, and packages. The state serves as the single source of truth for the backend simulation engine, ensuring deterministic execution and accurate synchronization with the frontend visualization.

The logistics network is modelled as a directed graph, where **Nodes** represent geographic locations and **Edges** represent traversable routes connecting them. Each node can host one or

more **Buildings**, such as depots or gas stations, which serve as operational points for trucks and packages. Specialized edge types, such as **Roads**, include traffic-related attributes like maximum speed or assigned trucks.

Dynamic entities, including **Agents**, **Trucks**, and **Packages**, interact continuously within this structure. Trucks act as mobile agents moving between nodes to deliver packages according to predefined or adaptive rules. Each building and vehicle maintains its own local state—such as current fuel level, stored packages, or active capacity—which collectively form the global simulation state. This state evolves over time through discrete events such as departures, arrivals, and loading operations.

All objects in the simulation are uniquely identified (via UUIDs) and linked through explicit references. This allows the backend to serialize, transmit, and reconstruct the simulation state efficiently during communication with the frontend through WebSocket messages.

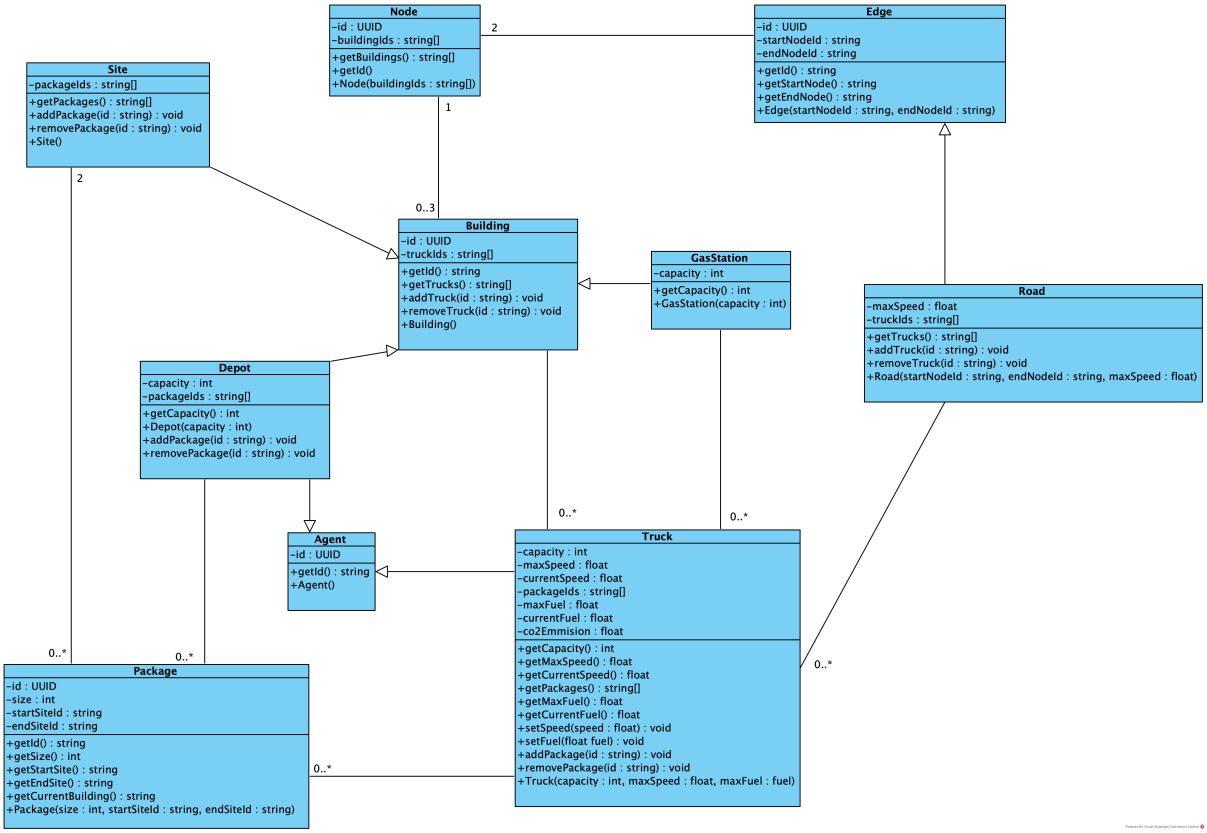

Figure 5: Class diagram representing the structure of the simulation state and relationships between core entities.

The structure illustrated on the figure 5 defines the conceptual foundation of the simulation model:

- **Static components** (nodes, edges, roads, and buildings) define the physical topology of the logistics network.
- **Dynamic components** (trucks, packages, and agents) evolve over time, representing operational activity.
- **Hierarchical relationships** allow specialization of entities (e.g., buildings as depots or gas stations, edges as roads).
- **Associations** between objects (e.g., trucks assigned to roads or depots) describe the runtime dependencies that drive simulation logic.

This unified state representation enables consistent management, serialization, and visualization of the simulation world, supporting real-time updates and synchronized rendering between backend and frontend components.

10 User Interface

Figure 6 presents a general vision for an user interface inside web browser's window. HUD elements like playback controls, controls manual and event log are displayed on top of a 3D, rendered scene.

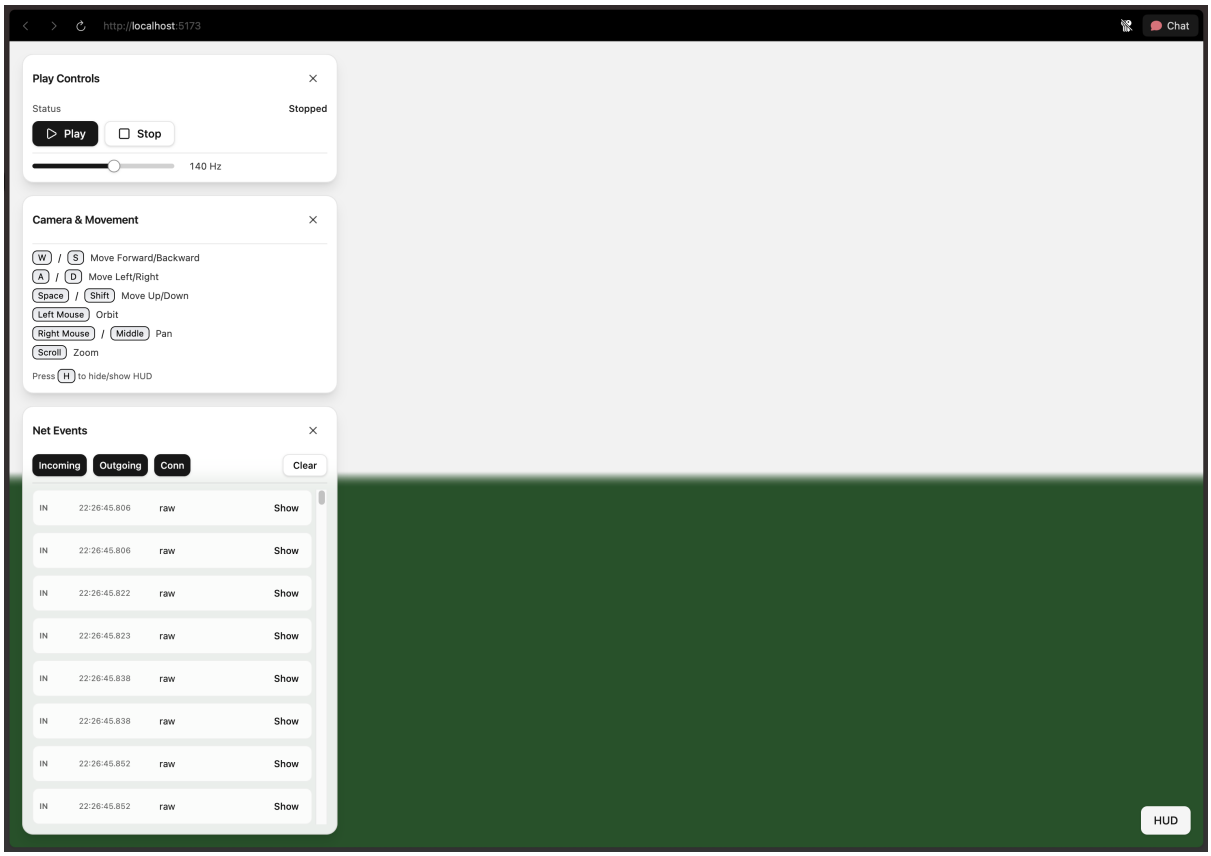

Figure 6: User interface vision - empty scene with visible HUD elements

11 Technology Selection

11.1 Frontend Technologies

11.1.1 Core Language and Type System

TypeScript TypeScript [4] was selected as the primary development language to provide static type checking and enhanced developer experience. The layered architecture benefits significantly from TypeScript's type system, which enforces the boundaries between network wire types, domain types, and view models. The strict compiler configuration maintains code quality and prevents common runtime errors. TypeScript's support for modern JavaScript features (ES2022 target) enables efficient code generation while maintaining compatibility with WebGL APIs and browser APIs.

11.1.2 Build Tools and Development Environment

Vite Vite [5] serves as the build tool and development server, chosen for its exceptional development experience and fast build times. The architecture's separation of concerns benefits from Vite's efficient module resolution and hot module replacement (HMR), allowing developers to modify individual layers without full rebuilds. Vite's native ES module support aligns with the project's use of ES modules throughout, and its plugin system enables seamless integration with React and TypeScript.

Bun Bun [6] is used as the JavaScript runtime and package manager for development tasks. Its fast package installation and execution speeds accelerate development workflows. While the production build targets standard browser environments, Bun's TypeScript support and compatibility with Vite enable a streamlined development experience without requiring separate transpilation steps for development scripts.

11.1.3 Rendering Technology

Three.js Three.js [1] was selected as the 3D rendering library for its mature ecosystem, comprehensive documentation, and widespread adoption. The library provides a high-level abstraction over WebGL that simplifies scene management, camera controls, and material systems while maintaining performance. Three.js's object-oriented design aligns well with the view layer's responsibility of managing mutable scene graphs, and its extensive geometry and material APIs support the project's low-poly aesthetic requirements. The library's active maintenance and large community ensure long-term support and availability of solutions to common rendering challenges.

11.1.4 User Interface Framework

React React [2] was chosen as the UI framework for the HUD layer due to its component-based architecture and declarative programming model. The HUD's modular design, where panels can be independently shown and hidden, maps naturally to React's component composition patterns. React's virtual DOM provides efficient updates for UI state changes, which is important when displaying real-time network telemetry and playback controls. The framework's extensive ecosystem and tooling support rapid development of complex UI components.

Radix UI Radix UI [7] provides headless, accessible UI primitives that form the foundation of the HUD components. Radix UI's composable primitives allow building complex components (dropdowns, dialogs, accordions) from simple building blocks, reducing code duplication and maintenance burden.

shadcn/ui shadcn/ui [8] is used as a component library built on Radix UI primitives. Rather than installing a traditional component library, shadcn/ui components are copied into the project, providing full control over styling and behavior.

11.1.5 Styling

Tailwind CSS Tailwind CSS [9] was selected as the utility-first CSS framework for its developer productivity and consistent design system. The framework's utility classes enable rapid UI development without writing custom CSS, which is particularly valuable for the HUD layer's numerous small components.

11.1.6 Runtime Validation

Zod Zod [10] was chosen as the schema validation library for its TypeScript-first design and runtime validation capabilities. The network layer uses Zod schemas to validate all incoming messages at the system boundary, protecting the domain layer from malformed data. Zod's type inference generates TypeScript types from schemas, ensuring that validation logic and type definitions remain synchronized.

11.2 Backend Technologies

11.2.1 Core Language and Dependency Management

Python (Poetry-managed) The backend is implemented in Python (`^3.10`) and managed with Poetry. The project enforces a strongly typed and quality-gated workflow via `mypy -strict`, `ruff` (lint + format), and comprehensive tests (pytest). All runtime and tooling versions are pinned in `pyproject.toml` to ensure reproducible builds and consistent CI behavior. Python was chosen for its ecosystem fit: the hierarchical map generator depends on mature scientific libraries (NumPy/SciPy), while the ASGI stack (FastAPI/Uvicorn) provides excellent async I/O for WebSocket traffic. Poetry was selected over `pip` + `virtualenv` to guarantee deterministic dependency resolution and simple environment bootstrapping for contributors and CI.

11.2.2 Web Framework and ASGI Runtime

FastAPI FastAPI [11] is used as the HTTP/WebSocket framework. Its async-first design, typing-friendly request/response models, and tight integration with Pydantic v2 make it a natural fit for our event-driven simulation backend. FastAPI powers the control endpoints and the WebSocket gateway used by the frontend. It was chosen over alternatives (Flask, Django) because it offers native async support, automatic OpenAPI generation for the API reference, first-class WebSocket routing, and zero-boilerplate dependency injection—all of which reduce glue code and improve correctness under strict typing.

Uvicorn Uvicorn [12] serves as the ASGI server. The `standard` extra enables performant event loop policies and production-ready defaults. Uvicorn's lightweight footprint and excellent FastAPI integration allow low-latency, high-throughput bidirectional communication. It was selected for its stable WebSocket implementation, small memory profile, and straightforward deployment story in containerized environments.

11.2.3 Data Modeling, Validation, and Serialization

Pydantic v2 Pydantic [13] v2 is used to define and validate backend domain messages. Models such as `Action` and `Signal` leverage field validators and post-validators to enforce contract correctness at the API boundary. We follow v2 idioms (`model_validate`, `model_dump`) and keep models fully typed. Pydantic was chosen because it unifies runtime validation and static typing, eliminating drift between schemas and code while offering high performance in v2.

orjson orjson [14] is configured as the JSON engine for fast, zero-copy serialization of signals and state snapshots. This reduces latency for large map payloads and frequent tick updates. It was selected over the standard library encoder for its significantly better throughput, correct handling of datetimes/NaNs, and compact output—all important for high-frequency WebSocket signals.

11.2.4 Scientific and Graph Tooling

NumPy and SciPy NumPy [15] and SciPy [16] power the hierarchical procedural generation pipeline (e.g., Poisson disk sampling, Delaunay triangulation, cKDTree queries). These libraries provide vectorized math and robust spatial algorithms needed to generate realistic road networks at scale. They were chosen because they are the de-facto standard for numerical computing in Python and offer well-tested implementations of the geometric primitives used by the generator.

NetworkX NetworkX [17] is available for analysis and utilities. While the runtime graph is represented by a custom lightweight structure optimized for simulation, NetworkX is used during testing and diagnostics when graph algorithms or conversions are convenient. This separation keeps the hot path minimal while retaining rich tooling for validation and research workflows.

11.2.5 Observability and Reliability

structlog structlog [18] provides structured, context-rich logging across the backend. Logs include action types, simulation ticks, and error contexts, which simplifies troubleshooting and supports downstream log aggregation. structlog was favored over ad-hoc logging because it encourages consistent JSON-shaped logs, which are easier to ingest into observability stacks.

11.2.6 Testing and Quality Gates

pytest, asyncio, benchmark, coverage The test stack uses pytest [20] with `pytest-asyncio` for async components, `pytest-benchmark` for microbenchmarks (e.g., generation timings), and `pytest-cov` to track coverage. The repository contains unit tests for generation, serialization, GraphML import/export, and simulation control. This stack was chosen for its expressive fixtures, fast developer feedback, and robust ecosystem of plugins.

mypy (strict) and ruff Type checking is enforced with mypy in strict mode; linting and formatting are enforced by ruff. This combination catches interface drifts early and keeps the codebase consistent and safe for refactors. Pre-commit hooks run these checks locally to mirror CI. Strict typing was chosen deliberately to make domain boundaries explicit and to future-proof the codebase as the simulation grows in scope.

11.2.7 Packaging and Execution

Poetry Poetry manages dependency resolution, virtualenvs, and scripts. The `pyproject.toml` declares both runtime dependencies (FastAPI, Uvicorn, Pydantic v2, orjson, NumPy, SciPy, NetworkX, structlog, prometheus-client) and development tools (pytest, mypy, ruff, coverage), reflecting the backend's requirements precisely. Poetry was selected because it simplifies onboarding (single `poetry install`) and produces deterministic lockfiles suited for CI/CD.

11.2.8 Architecture Alignment

Event-driven design The backend follows an event-driven model built around typed *Actions* and *Signals* (see `world/sim/queues.py`). Enumerated types map to canonical wire strings (e.g., `map.created`), and Pydantic models validate messages at the boundary. This, combined with ASGI and WebSockets, enables high-frequency updates (ticks, site statistics, package lifecycle) without blocking the generator or simulation loop. The selection of FastAPI/Uvicorn/Pydantic/orjson directly supports this architecture: async I/O for concurrency, strict schemas for safety, and fast serialization for throughput.

11.3 Communication Technologies

WebSocket The native browser WebSocket API [3] is used for bidirectional communication with the SPINE backend. WebSocket was selected over alternatives (Server-Sent Events, HTTP polling) because it provides low-latency, full-duplex communication necessary for real-time simulation updates. The protocol's ability to push updates from server to client eliminates the need for polling, reducing network overhead and enabling responsive user interactions. WebSocket's event-driven model aligns with the architecture's event-driven state management pattern, where incoming messages trigger domain events that update simulation state.

12 Bibliography

References

[1] Three.js documentation. `https://threejs.org/docs/`
Accessed 16.10.2025

[2] React Learn. `https://react.dev/learn`
Accessed 16.10.2025

[3] MDN Web Docs: WebSockets API. `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`
Accessed 16.10.2025

[4] TypeScript documentation. `https://www.typescriptlang.org/docs/`
Accessed 01.11.2025

[5] Vite documentation. `https://vite.dev/guide/`
Accessed 01.11.2025

[6] Bun documentation. `https://bun.com/docs`
Accessed 01.11.2025

[7] Radix UI documentation. `https://www.radix-ui.com/themes/docs/overview/getting-started`
Accessed 01.11.2025

[8] shadcn/ui documentation. `https://ui.shadcn.com/`
Accessed 01.11.2025

[9] Tailwind CSS documentation. `https://tailwindcss.com/docs/installation/using-vite`
Accessed 01.11.2025

[10] Zod documentation. `https://zod.dev/`
Accessed 01.11.2025

[11] FastAPI documentation. `https://fastapi.tiangolo.com/`
Accessed 02.11.2025

[12] Uvicorn documentation. `https://www.uvicorn.org/`
Accessed 02.11.2025

[13] Pydantic v2 documentation. https://docs.pydantic.dev/latest/
Accessed 02.11.2025

[14] orjson documentation. https://github.com/ijl/orjson
Accessed 02.11.2025

[15] NumPy documentation. https://numpy.org/doc/
Accessed 02.11.2025

[16] SciPy documentation. https://docs.scipy.org/doc/scipy/
Accessed 02.11.2025

[17] NetworkX documentation. https://networkx.org/documentation/stable/
Accessed 02.11.2025

[18] structlog documentation. https://www.structlog.org/en/stable/
Accessed 02.11.2025

[19] prometheus-client documentation. https://github.com/prometheus/client_python
Accessed 02.11.2025

[20] pytest documentation. https://docs.pytest.org/
Accessed 02.11.2025

[21] pytest-asyncio documentation. https://pytest-asyncio.readthedocs.io/
Accessed 02.11.2025

[22] pytest-benchmark documentation. https://pytest-benchmark.readthedocs.io/
Accessed 02.11.2025

[23] pytest-cov documentation. https://pytest-cov.readthedocs.io/
Accessed 02.11.2025

[24] mypy documentation. https://mypy.readthedocs.io/en/stable/
Accessed 02.11.2025

[25] Ruff documentation. https://docs.astral.sh/ruff/
Accessed 02.11.2025

[26] Poetry documentation. https://python-poetry.org/docs/
Accessed 02.11.2025