Universität Rostock

Traditio et Innovatio

Transforming Graph Models of Smart Environments for Simulation Purposes

Name:	Maneshi
Vorname:	Amin
Matrikel-Nr.:	221202545
Studiengang:	Master of Science Informationstechnik / Technische Informatik
E-Mail:	a.maneshi@uni-rostock.de
Fakultät:	Faculty of Computer Science and Electrical Engineering
Betreuer :	Florian Rose, MSc
1. Gutachter :	Florian Rose, MSc
2. Gutachterin :	Dr.-Ing. Anke Dittmar
Abgabedatum:	26.05.2025

Master Thesis

University of Rostock

CONTENTS

List of Figures

List of Tables

1 Introduction

Smart technologies are becoming part of everyday life. From traffic lights that adjust automatically to real-time road conditions, to sensors that help manage energy use in buildings, smart environments are reshaping how we address technical challenges and societal needs. These systems are powered by networks of connected devices, such as sensors and small computers, that collect and share data to support automated decisions. According to the European Commission, the number of connected devices worldwide is expected to surpass 55 billion by 2025 [1], showing just how quickly this field is growing.

Smart environments are used in many different areas, including transportation, healthcare, home automation, and industry. A traffic control system might use sensors and cameras to manage vehicle flow, while factories rely on sensor networks to monitor machines and ensure safety. One of the biggest challenges across these settings is getting all these devices to work together smoothly. As systems grow more complex, setting up and testing them manually becomes harder and slower. Even a small system like a smart intersection can require many steps, defining databases, writing code, and setting up communication between parts. These efforts only increase as the system gets larger.

To model and design such systems, graph-based representations are often used. In these models, each node stands for a physical device or logical component, while the edges show how data flows or which parts communicate. This approach is flexible and easy to understand. Previous work, like that of Hemel et al.(2010) [2], showed how structured models can help generate code [2], and others like Fogh et al.(2010) [3] applied similar ideas to specialized domains like bioinformatics [3]. Still, there's a missing link: going from a graph-based model to a fully working simulation often involves a lot of manual work, especially when databases, data handling, and network behavior must be included. This slows down development and makes experiments harder to repeat.

1.1 Motivation and Research Gap

Even though many tools and frameworks exist for designing smart environments, most of them only focus on one part of the process. For example, Yao and Gehrke(2003) [4] explored how to process queries in sensor networks, but didn't cover the full pipeline from model to simulation [4]. Similarly, the MEMOPS project [3] focused on generating code from models but was limited to specific applications.

Because of this, researchers and developers often have to build their simulations by combining many tools manually. This takes time, increases the chance of mistakes, and makes it harder to compare results or reuse previous work. In areas where fast testing and flexibility are important, like traffic control or emergency planning, this is a serious problem.

What's missing is a system that takes a complete model and automatically turns it into a simulation that's ready to run. This includes setting up databases, generating the right code, configuring behavior, and deploying everything across separate nodes. Such an automated pipeline would make it easier to design, test, and improve smart environments without spending hours on setup.

1.2 Research Goals and Methodological Approach

The main goal of this thesis is to build a fully automated transformation pipeline that turns abstract graph-based models of smart environments into simulations that are executable. This means covering all the steps from reading the input model to deploying code and setting up databases on different nodes, without needing to write everything manually.

The idea builds on model-driven engineering (MDE) [2, 5], but goes further than traditional code generation. The pipeline also handles database setup, behavior logic, and communication between nodes in real time. In smart systems, it's not enough for each node to work alone, they often need to ask other nodes for data. To support this, the pipeline allows nodes to send HTTP-based queries to each other. For example, one node might ask a neighbor if an emergency vehicle was detected. The answer, drawn from the target node's database, helps the system respond in a realistic and decentralized way.

This query-driven design makes the simulation more flexible. Instead of just sending messages blindly, each node can look at current conditions, reason about what to do, and adjust its behavior accordingly. All of this is controlled by configuration files and JSON schemas, so the setup remains simple and reusable.

The pipeline was built using a modular approach. Each part, input processing, schema generation, behavior setup, and deployment, was developed and tested separately, then brought together into a full working system. To demonstrate what it can do, the system was tested with two examples: one simulating a smart traffic intersection, and another representing a smart home with rooms that sense and share environmental data.

In both cases, the pipeline was able to automatically create the needed components: virtual machines (VMs) for each node, local databases, communication logic, and runtime behavior. It worked reliably and quickly, showing that the approach is practical for real-world applications like smart cities, home automation, and industrial IoT [6, 7].

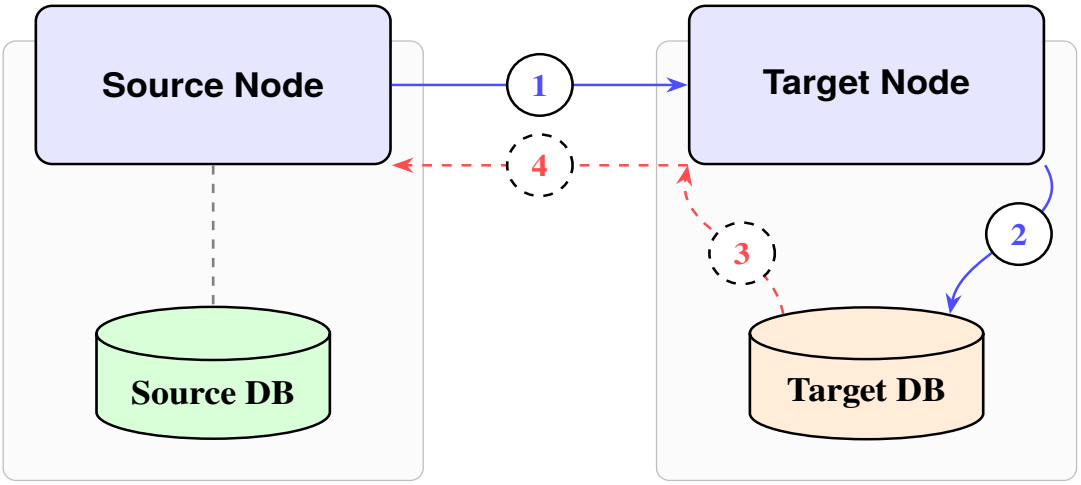


Figure 1: Inter-node query-response flow showing how a node retrieves real-time data from a neighbor. The source node sends a request (1), the target queries its database (2), returns the result (3), and sends the response (4).

1.3 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 provides background information, a review of related work, and an overview of the technological stack, including theoretical foundations and existing tools. Chapter 3 presents a detailed overview of the transformation pipeline, covering input configuration, schema-based data generation, graph modeling, virtual deployment, microservice design, and communication mechanisms. Chapter 4 introduces the evaluation methodology and presents two case studies: an urban traffic management scenario and a smart home environment, followed by an analysis of results. Chapter 5 discusses the broader implications of the work, identifies limitations. Finally, Chapter 6 concludes the thesis and suggests future directions.

2 Background, Related Work, and Technological Stack

This chapter introduces the core ideas and technologies behind the transformation pipeline developed in this thesis. It first presents the theoretical background, including graph-based modeling, MDE, and distributed database schema design. These areas help explain how smart environments can be formally described, modularly structured, and automatically converted into simulation-ready systems. The chapter reviews related research to highlight current gaps in automation, coordination, and scalability. This context sets the stage for the integrated, schema-driven pipeline presented in the rest of the thesis. Finally, the chapter surveys existing technologies and tools commonly used in modeling, deployment, and simulation, from graph libraries like NetworkX to database systems and virtualization platforms. While many of these tools are powerful on their own, they often operate in isolation and require manual integration.

2.1 Theoretical Foundations

To support the proposed system, this section presents the theoretical building blocks that shape its structure. It begins with graph modeling, which provides the foundation for defining smart environments. It then explores principles from MDE, which guide the system's automation, followed by the design of distributed database schemas, which enable simulation nodes to function independently. Finally, it introduces the concept of translating abstract models into executable behavior, showing how high-level definitions are systematically mapped to operational logic across a network of nodes..

2.1.1 Graph-Based Modeling in Smart Environments

Graph-based representations offer a powerful and intuitive framework for modeling systems composed of many interacting components. At a basic level, a graph consists of nodes (such as sensors, actuators, or controllers) and edges that represent the relationships or data flows between them [8]. This abstraction is particularly relevant for smart environments, where numerous devices communicate in structured and dynamic ways.

Directed graphs are especially useful in this context, as they capture the directionality of communication, clarifying which components send data and which receive it [9]. Building upon this, property graphs extend the basic graph model by allowing both nodes and edges to hold descriptive attributes. For instance, a node might indicate the type or role of a device, while an edge could define the format or context of the transmitted data [7].

In more complex scenarios, where multiple types of communication or redundancy exist between the same components, multigraphs become valuable. These allow multiple labeled edges between a single pair of nodes, enabling the representation of different communication channels or message types [6].

By combining directed graphs, property graphs, and multigraphs, it becomes possible to construct rich and adaptable models of smart environments, models that reflect not only the structural layout of devices but also the behavior, roles, and data interactions within the system.

2.1.2 Principles of Model-Driven Engineering

MDE is a well-established software development approach in which high-level models serve as the primary artifacts for defining system logic and architecture. Instead of coding every system detail manually, developers define models that describe structure and behavior, and rely on tools to generate

executable components automatically [10]. This allows for faster development and reduces the likelihood of inconsistencies between design and implementation [11].

MDE is particularly valuable in the context of smart environments, where systems are often distributed and composed of heterogeneous devices. It supports abstraction, minimizes repetitive implementation tasks, and enables developers to focus on domain functionality rather than infrastructure [12]. When models are used as the central reference point, consistency between components is improved, which is especially important in systems where multiple services and devices interact [13].

Unified Modeling Language (UML) is one of the most widely adopted modeling languages in MDE. It offers standardized visual notations for defining system structure (e.g., class diagrams) and behavior (e.g., activity or state machine diagrams) [14]. UML supports both abstract (platform-independent) and concrete (platform-specific) modeling, making it suitable for a broad range of applications [15].

For more domain-specific applications, however, many systems benefit from the use of Domain-Specific Languages (DSLs). DSLs are custom modeling languages tailored to specific domains, enabling subject-matter experts to express behavior and system logic more naturally [16]. In smart environments, a DSL might be used to describe sensors, data types, communication rules, or node behaviors.

In this thesis, JSON Schema is used as a lightweight and machine-readable DSL [17]. Each schema defines the data structure, behavior expectations, and interaction logic of a node. The transformation pipeline uses these schemas to automatically generate simulation data, define data exchange structures, and support query handling logic across nodes. This schema-driven approach ensures consistency, simplifies configuration, and enables scalable deployment without manual implementation of low-level behaviors.

2.1.3 Designing Schemas for Distributed Simulation

In the simulation system developed in this thesis, there is no central database. Instead, each node maintains its own local database and handles its data independently. This decentralized approach mirrors how real smart environments often function. Processing data close to where it is generated allows for faster decisions, reduces network dependency, and supports modular system design [18].

Since each node is fully responsible for storing and managing its own data, the system does not rely on traditional database normalization techniques. While normalization is often used in centralized databases to reduce redundancy, in distributed simulations, selective duplication (or denormalization) can be more practical. For instance, a node may temporarily store a summary of a neighbor's data to avoid repeated queries and reduce latency [19]. This improves performance without compromising independence.

Rather than splitting a large dataset across machines, as is common in conventional distributed databases [19], this simulation embraces a naturally fragmented structure [18]. Each node holds only the data it collects or receives from others. For example, a smart home node managing the kitchen stores only kitchen-related sensor values. This structure makes the system simpler, more efficient, and easier to scale [20].

Because smart environments frequently change (devices may be added, removed, or updated), the use of JSON Schema allows for flexible validation and supports optional fields. This makes it easy to adapt the database structure at each node without breaking the overall system [21].

Much of the data handled in these systems is time-based, such as periodic readings from motion detectors, CO_2 sensors, or light monitors. Each node stores time-stamped records locally, enabling it to analyze patterns or respond to recent events. Indexing and efficient queries help ensure that these

operations remain fast and responsive [22].

Overall, the simulation architecture adopts an edge-first design philosophy, where all data storage and decision-making occur at the node level. This decentralized model reflects how modern smart environments increasingly rely on distributed processing to reduce latency, enhance resilience, and support real-time responsiveness. Although the current implementation does not integrate with cloud-based systems, its schema-driven structure allows for seamless future extensions. For instance, synchronizing selected data with a remote dashboard or cloud service could be introduced without disrupting the autonomous behavior of individual nodes [20].

2.1.4 From Models to Behavior

A key contribution of this thesis is the ability to transform high-level system models into executable, distributed simulations. The pipeline begins with a graph-based configuration, including nodes, edges, and metadata (such as node types, schema references, or communication roles), and generates a working system in which each node operates as an independent Flask-based service with its own local database.

Rather than hardcoding behavior into each node, the system uses schema definitions to determine what data each node handles and how it should react. These schemas drive the structure of each node's logic, defining expected fields and shaping query behavior during runtime. For example, if a node's schema includes fields such as `light_level` or `motion`, its query interface will reflect those attributes, enabling selective, data-driven responses throughout the simulation.

This model-driven approach ensures that the system behaves as the designer intended. The underlying concept aligns with work by Morin et al.(2009) [13], who highlight how model-based engineering improves consistency and automation across distributed systems. [13]. Voelter (2013) [16] emphasizes the use of domain-specific languages to capture behavior-specific logic in structured formats, an idea reflected here in the use of JSON Schema [16]. Vogel et al.(2014) [23] explore the direct execution of behavioral models, reinforcing the idea that simulation logic can be derived from abstract model definitions [23]. Finally, Pezoa et al.(2016) [21] describe the foundational principles of schema-driven data handling, which support the automatic interpretation and validation of structured input [21].

This ensures the simulation behaves as designed, while remaining modular and easy to adapt without extensive manual effort.

2.2 Related Work

This section reviews existing research relevant to the development of an automated simulation pipeline for smart environments. The literature is organized into four areas: graph-based modeling, automated code generation, schema generation for databases, and simulation tools. Together, these studies highlight how far the field has progressed, and where gaps remain.

2.2.1 Graph-Based Modeling Approaches

Graphs are a widely adopted abstraction for modeling smart environments because they effectively represent interactions between distributed components. In such models, nodes correspond to entities like sensors, controllers, or services, while edges define communication flows or dependencies [8]. This structure maps well onto IoT applications, where physical and logical connectivity are equally important.

Fortino et al. (2018) [24] proposed a graph-based IoT modeling framework that not only captures system topology but also embeds behavioral logic at each node. This approach is conceptually aligned with this thesis, which also treats nodes as logic-bearing units. However, Fortino's work focuses primarily on modeling and analysis, whereas the pipeline introduced here extends this concept into deployment by automatically generating running simulations from graph definitions [24].

Morin et al. (2017) [25] developed a dynamic, runtime-adaptive graph structure where the topology evolves to reflect changes in the environment. Their contribution emphasizes runtime flexibility, an important consideration for real-world IoT systems. Although this thsis does not support dynamic reconfiguration, it shares the goal of enabling modularity and traceability through high-level models [25].

Gyrard et al. (2015) [26] introduced semantic graphs that incorporate ontologies for cross-domain interoperability. This is particularly important for large, heterogeneous environments. While semantic integration is not a primary focus of this thesis, the use of schemas for defining node behavior and communication mirrors their intent of making device behavior interpretable and consistent across contexts [26].

While these studies highlight the versatility of graph-based modeling for IoT systems, they often focus on conceptual modeling, semantic alignment, or centralized planning. In contrast, the contribution of this thesis lies in bridging high-level graph models and real, executable simulations. It moves beyond static design by automating deployment, node behavior, and runtime coordination based on model input, providing a lightweight, practical tool for testing smart environments at execution time.

2.2.2 Automated Code Generation Techniques

One of the main goals of model-driven development, as described by Schmidt (2006) [10], is to reduce the need for writing low-level code by hand. Instead, developers can work with high-level models that describe how the system should work, and let tools handle the code generation. This makes development faster and helps keep the design consistent[10].

This idea is reflected in the pipeline developed for this thesis. Rather than manually coding each part of the system, the pipeline starts from a model (a graph made of nodes, edges, and schemas) and automatically sets up the simulation. Each node becomes a running Flask service with its own database, and communication between nodes is configured based on the model. This approach makes it easier to test changes, add new parts, or reuse existing setups without repeating the same work every time.

Hemel et al. (2010) [2] proposed a modular transformation approach that separates code generation into reusable blocks, improving adaptability and maintainability. While their work does not address simulation behavior directly, the idea of modular generation inspired the structure of this thesis's transformation pipeline, which separates graph processing, schema-driven data generation, and simulation deployment into clearly defined phases.

Prehofer and Chiarabini (2015) [27] worked on generating efficient code for embedded IoT devices, with a focus on edge computing. Their approach is well-suited for single devices, but it doesn't cover how multiple nodes work together or how to run a full simulation across a network[27].

Brambilla et al. (2017) [12] created a toolchain that turns IoT models into REST APIs and web interfaces. This helps with quick prototyping and building user interfaces, but it doesn't handle simulation logic or setting up back-end systems across multiple nodes[12].

Ciccozzi et al. (2013) [28] worked on turning UML models into control code for real-time systems. Their framework handles precise timing well, but it's designed for single devices, not for running

distributed simulations across multiple services[28].

In contrast, the transformation pipeline developed in this thesis expands traditional MDE principles by enabling the automated construction of distributed, schema-driven simulation environments. It uses JSON Schema [17] to define the structure and expected behavior of each node. From these definitions, the system automatically generates simulation data structures and data exchange logic, which are then deployed via lightweight Flask-based services that communicate over HTTP [29]. While the REST interfaces themselves are not generated in full, they are dynamically populated and shaped by the schema definitions, allowing runtime adaptability. The system architecture reflects the principles of decentralized, edge-oriented computing [20], with each node acting independently while maintaining synchronized interactions through schema-informed messaging.

2.2.3 Database Schema Generation

Turning abstract models into working database schemas is an important step in simulating real-world systems. This process ensures that each component can store and access data in a way that reflects the system's design and behavior.

Demuth et al. (2001) [30] presented one of the early methods for generating relational database schemas directly from UML class diagrams using the Object Constraint Language. Their approach ensured that structural rules and constraints defined in the model were also enforced in the database. However, their work focused on static systems and did not address the type of time-sensitive or sensor-based data commonly found in smart environments [30].

To support more flexible data handling, Mehmood et al. (2017) [31] proposed a method for turning high-level models into MongoDB schemas. Their work fits well with NoSQL databases, especially when the data structure can change over time. This idea is useful for real-time systems, and it relates to this thesis, where JSON Schema is used to define data formats for each node. Like their approach, it allows the system to work with evolving data without needing to redesign the whole database [31].

From a practical database management perspective, Schwartz et al. (2012) [32] provided useful advice on improving MariaDB performance. They covered things like indexing, memory settings, and storage engines like InnoDB, which are important for systems that need fast, concurrent access to data[32]. Bell (2018) [33] built on this by showing how MariaDB can be a good fit for IoT applications, thanks to its small size and ability to run on low-power devices at the network edge[33].

This thesis builds on these ideas by using JSON Schema to define the structure of each node's data in the simulation. These schemas describe the fields, data types, and expected content, and are used to configure MariaDB databases locally on each VM. While the system does not automatically generate full SQL schemas, it uses the JSON definitions to guide how each node stores and handles its data. This includes support for time-stamped sensor readings and schema-informed updates. By combining schema-driven design with a lightweight MariaDB setup, the system ensures that each node operates independently but still follows a consistent structure, aligned with the goals of flexible, edge-based simulation.

2.2.4 Simulation Environments for Smart Systems

Simulation plays a key role in testing distributed smart systems before deployment. It allows developers to safely evaluate system behavior, performance, and resilience under different conditions, without relying on real-world infrastructure.

Zeng et al. (2017) [34] introduced IoTSim, a detailed simulator for IoT systems with support for modeling devices, networks, and cloud resources. However, its manual setup process and lack of schema integration make it less suitable for rapid prototyping or flexible, automated deployment (key goals of this thesis.)[34].

Santana et al. (2017) [35] developed CityPulse, a smart city simulation platform that integrates live and historical data for urban analysis. Their work inspired the multi-domain applicability of this thesis, but their system is focused on centralized urban infrastructure, whereas the system presented here is designed for decentralized environments like smart homes or edge-based networks [35].

Khan et al. (2013) [36] proposed a cloud-based IoT testing framework focused on communication protocols. While their work highlights protocol-level validation, it does not address high-level behavior modeling or automated configuration of simulation logic across distributed services, which this thesis attempts to solve through schema-driven setup[36].

Corbett et al. (2013) [37] emphasized the importance of fault tolerance in distributed systems, using techniques like fault injection and stress testing to ensure reliability. This idea influenced the design of the simulation in this thesis, where each node is deployed as an independent service with its own database and logic. If one node fails or restarts, it can recover without affecting the rest of the system. (supporting resilience without relying on a centralized controller.)[37].

Malinowski et al. (2007) [38] introduced MacroLab, a framework that simplifies programming in sensor networks by allowing centralized scripts to be compiled into distributed implementations. This approach is useful for understanding how logical behavior can be expressed at a high level and mapped to distributed systems. In the context of this thesis, MacroLab's abstraction of distributed behavior inspired the separation between global system modeling and node-level execution. However, unlike MacroLab, this thesis focuses on simulation rather than deployment code generation, and emphasizes schema-driven configuration, local database handling, and real-time query support across independently operating nodes[38].

Jensen et al. (2017) [39] explored efficient storage and querying of time-series data in sensor-driven systems. While their work does not focus on simulation, it helped shape the local database design in this thesis, where each node stores time-stamped sensor data. This enables realistic tracking of environmental changes, supports time-based queries, and allows each node to analyze recent events independently [39].

Building on these contributions, this thesis proposes a lightweight, schema-driven simulation environment tailored for smart environments. It supports automatic deployment of distributed nodes, real-time behavior modeling, and flexible adaptation to various domains. By using JSON Schema to define node logic and communication, and by deploying each node as a modular Flask service, the system enables scalable, reusable simulations with minimal setup, addressing gaps in usability, decentralization, and domain flexibility found in earlier work.

2.2.5 Research Gap

While many tools exist to support specific aspects of system modeling or simulation, few provide a complete solution that spans modeling, deployment, and runtime behavior. Code generation frameworks like MEMOPS [3] produce efficient embedded code but do not support dynamic simulation, runtime queries, or modular deployment. Schema tools focus primarily on static validation and storage formatting, without connecting model semantics to actual runtime execution. Similarly, simulation platforms such as IoTSim [34] and CityPulse [35] support large-scale testing and data replay but often depend on centralized

architectures and fixed configurations, limiting adaptability and schema reuse.

MacroLab [38] offers a higher-level abstraction for sensor network programming, compiling centralized scripts into distributed behavior. However, it lacks real-time query support, local database interaction, and modular simulation control, features addressed by this thesis. Spanner [37], on the other end of the spectrum, exemplifies large-scale, fault-tolerant infrastructure optimized for distributed database systems. While it offers resilience and high performance, it is not designed for schema-driven simulation or modular behavior modeling, and does not support configuration-based behavior execution.

As summarized in Table 1, these systems only partially meet the needs of schema-driven, automated simulation pipelines, particularly in decentralized smart environments.

Approach	Code Gen.	Simulation	Smart Env. Support	Automated
MacroLab [38]	Partial	✗	✓	✗
MEMOPS [3]	✓	✗	✗	✗
Spanner [37]	✗	✓	✗	Partial
This Pipeline	Schema-Driven	✓	✓	✓

Table 1: Comparison of related work with the proposed transformation pipeline.

In response to these limitations, this thesis presents a lightweight, fully automated transformation pipeline that begins with high-level JSON-based graph models and generates operational, decentralized simulations. It integrates modeling, deployment, database setup, and runtime behavior without requiring manual coding or scripting. By aligning data structure, communication, and behavior through schema definitions and declarative configuration, the system supports simulation of smart environments in a scalable, modular, and reusable way.

2.3 Existing Technologies and Tools

This section reviews current technologies and tools relevant to modeling, simulating, and deploying smart environments. While many of these tools are powerful and widely used, they often handle only part of the overall process, creating gaps that this thesis addresses through an integrated pipeline.

2.3.1 Graph Modeling Tools

Graph modeling tools play an important role in designing, analyzing, and visualizing network structures. In the context of smart environments, such tools help researchers and developers represent how devices are connected, how they interact, and how data flows between them. Among these tools, NetworkX is a widely used Python library for constructing and analyzing graph-based structures [40]. It supports various types of graphs, including directed, undirected, and multigraphs, and offers a wide range of built-in algorithms for tasks such as pathfinding, clustering, and centrality analysis. Thanks to its seamless integration with Python, NetworkX is commonly used in research environments for prototyping and experimenting with network topologies [41]. While NetworkX provides functionality for exporting graphs to standard formats such as GraphML, GML, JSON, and edge lists [42], these exports primarily serve purposes

like visualization, data sharing, or external processing. It does not, however, support direct generation of simulation logic, executable code, or database schemas from graph structures. Additionally, it lacks native features for persistent graph storage, version tracking, or managing very large graphs, which may limit its usefulness in large-scale or long-running system simulations.

In addition to NetworkX, several specialized graph visualization tools offer robust interactive capabilities. Gephi [43], Cytoscape [44], and D3.js [45] are widely used for exploring and presenting complex network data. These platforms help users identify structural patterns, clusters, or anomalies through intuitive visual interfaces, making them valuable for data-driven systems and smart environment applications. However, their primary focus lies in data analysis and presentation, and they do not integrate directly with simulation workflows, runtime environments, or code generation pipelines. As such, while powerful for visual exploration, their utility is limited when it comes to the end-to-end development of distributed smart systems.

2.3.2 Virtualization and Remote Management Tools

Deploying distributed simulations requires tools that can manage multiple VMs and automate remote execution. This section outlines the core technologies used to build, configure, and control the virtual nodes within the simulation pipeline, highlighting both their strengths and limitations. VirtualBox is a widely adopted open-source platform for creating and running VMs. It includes a robust command-line interface, `VBoxManage`, which enables detailed control over VM provisioning, hardware configuration, network setup, cloning, and snapshots [46]. These capabilities make it a practical choice for simulating distributed environments in a controlled and repeatable way. In this thesis, each virtual node is instantiated from a lightweight Debian-based image, chosen for its stability, minimal resource usage, and compatibility with tools such as Python and MariaDB [47]. While VirtualBox is flexible, configuring a network of VMs with static IPs and consistent software environments requires nontrivial scripting. Although `VBoxManage` supports automation, it is not designed for large-scale orchestration or dynamic scaling.

For remote access and automated control, Paramiko is used to manage SSH connections and execute commands programmatically [48]. It facilitates communication with each VM in the network, enabling operations such as file transfers and the launching of services like Flask APIs and MariaDB instances without manual intervention. Although Paramiko is reliable and well-suited to experimentation, it operates at a low level, requiring manual handling of session state, command sequences, and error recovery. As the number of virtual nodes grows, managing interactions becomes increasingly complex, suggesting that higher-level orchestration tools may be more appropriate for larger-scale deployments. Additionally, the Unix `nohup` utility is used to run services in the background after SSH sessions close, ensuring persistence of Flask servers and other processes [49]. While simple and lightweight, `nohup` lacks features for monitoring or automatic recovery, making it suitable for basic setups but limited in robustness compared to tools like `systemd`, `supervisord`, or container-based alternatives.

2.3.3 Network Configuration and Virtualization

Simulating distributed smart environments requires careful handling of network communication between virtual nodes. Each node must operate in isolation while still maintaining connectivity with its neighbors, reflecting how real-world systems exchange data. One common method used is Network Address Translation (NAT), which allows multiple VMs to share a single host IP address [50]. NAT is particularly effective in virtualized environments where numerous VMs coexist on the same physical machine.

However, it presents challenges for direct peer-to-peer communication, as internal addresses are hidden behind the host IP. To overcome this, additional configuration is needed, such as static port mapping or SSH tunneling, to ensure reliable messaging between nodes. In this thesis, NAT is combined with predefined port forwarding rules to allow each VM to receive HTTP requests from other nodes during the simulation.

Beyond basic address translation, modern network virtualization tools like Docker [51], Kubernetes [52], and Software-Defined Networking platforms [53] provide advanced methods for emulating realistic network topologies. These tools support fine-grained control over communication rules, latency, and traffic flow, making them valuable for modeling conditions such as multi-hop routing, sensor-to-cloud messaging, and failure injection. Despite their strengths in network orchestration, these technologies do not address other key aspects of simulation, such as schema generation, behavior modeling, or data synthesis. Therefore, in comprehensive smart system simulations, they are best used alongside complementary components like model transformation pipelines and scripting frameworks to support full end-to-end deployment and evaluation.

2.3.4 REST APIs and HTTP Communication

HTTP-based communication plays a central role in the simulation pipeline developed in this thesis. Each virtual node hosts a lightweight RESTful API to receive data, respond to queries, and coordinate with other nodes in real time. This communication is implemented using Flask [54], a minimalist Python web framework well-suited for small-scale services. The system launches a Flask server on each VM, exposing predefined endpoints for inter-node messaging. These services are started remotely via SSH using background execution tools like `nohup`, allowing them to persist independently of the controlling session [49]. Communication follows the HTTP/1.1 specification [55], using JSON payloads and standard request methods.

While Flask offers flexibility and simplicity, much of the supporting functionality (such as route definitions, input handling, and error management) was implemented manually in the codebase. This logic is independent of the pipeline's input: it is static code written once and reused across all nodes, regardless of the scenario configuration. These behaviors are not generated dynamically from the JSON-based models, but rather included as part of the reusable simulation service. As a result, although communication setup and deployment are automated by the pipeline, the internal API logic reflects additional static coding effort rather than runtime generation. In comparison, modern frameworks like FastAPI [56] offer built-in validation, automatic OpenAPI documentation, and native support for asynchronous execution, which could streamline future development and improve maintainability. FastAPI also integrates with Python's type hints and the Pydantic model system [57], reducing runtime errors and simplifying the addition of new endpoints. Despite these advantages, Flask was selected for this project due to its minimal dependencies, its compatibility with SSH-based remote deployment, and its alignment with the lightweight, modular goals of the simulation pipeline.

2.3.5 Data Formats and Schema Definitions

Structured data formats are critical in distributed simulations, where each node must interpret and validate its data consistently and independently. In this thesis, JSON is used as the primary data exchange format due to its lightweight syntax, readability, and compatibility with Python and web technologies [58]. To enable schema-driven automation, JSON Schema is employed as a formal mechanism for defining the

expected structure, data types, required fields, and validation constraints of each node's data model [17]. Within the simulation pipeline, these schemas serve as configuration blueprints: they are parsed automatically to generate database tables, construct sensor payloads, and inform inter-node communication behavior. This allows each virtual node to be provisioned and initialized based on a shared but flexible configuration model, while still enabling localized autonomy.

Although alternatives such as XML and XML Schema (XSD) provide similar capabilities for data validation and structure definition [59, 60], they were not chosen for this work. XML offers greater expressiveness, including namespace support and complex nested hierarchies, but this comes at the cost of increased verbosity and processing overhead [61]. In resource-constrained or performance-sensitive environments, such as distributed simulations running on VM, JSON provides faster parsing, simpler syntax, and easier programmatic handling. These advantages, combined with native support in Python and seamless integration with other pipeline components, make JSON and JSON Schema more suitable for dynamic, schema-driven system deployment in smart environments.

Feature	JSON / JSON Schema	XML / XML Schema
Readability	High (Concise and human-readable syntax) [58]	Moderate (Verbose tags reduce clarity) [61]
Parsing Speed	Fast (Simple structure allows quick parsing) [58]	Slower (Requires more processing due to nesting and metadata) [61]
Syntax	Simple, flat (Key-value pairs; minimal structure) [17]	Verbose, nested (Rich but complex markup hierarchy) [60]
Schema Validation	Supported (JSON Schema provides structure and constraints) [17]	Supported (XSD defines types, constraints, and nesting) [60]
Web/IoT Use	Common and flexible (Widely adopted in REST APIs and IoT platforms) [17]	Declining (Less common in modern web services) [59]
Python Integration	Native and easy (Handled directly via built-in `json` module) [17]	Requires external libraries (Parsing XML often needs tools like `lxml`) [59]
Namespaces	Not supported (No native namespace management) [17]	Supported (XSD allows full namespace handling) [60]
Edge Suitability	High (Lightweight and low-overhead) [58]	Low (Processing and bandwidth overhead are unsuitable for edge) [61]

Table 2: Comparison of JSON and XML

2.3.6 Database Management Systems

Selecting an appropriate database system is a critical aspect of designing a distributed simulation. In this thesis, each virtual node manages its own local database instance to store and query sensor data. The system must support efficient access, schema validation, and adaptability to evolving models. Three broad categories of database technologies (relational, NoSQL, and time-series) were evaluated based on their suitability for simulation environments and schema-driven development.

MariaDB, a widely used open-source relational database, was ultimately chosen. It supports full SQL querying, indexing, and ACID-compliant transactions [62], and is compatible with MySQL while offering reliable performance for structured data. In the simulation pipeline, MariaDB is deployed on each node to manage sensor values. Its fixed-schema model ensures consistency, and the database structure is guided by JSON Schema definitions [17], reducing manual setup. Alternative approaches include time-series databases such as InfluxDB and TimescaleDB. These systems are designed to efficiently store and query chronological data, making them popular in monitoring and IoT scenarios [63, 64]. TimescaleDB, in particular, enhances PostgreSQL with time-series indexing, combining performance with a familiar SQL interface. While effective at handling frequent inserts and time-window queries [65], time-series databases generally do not natively support schema-driven data models or automatic behavior generation, limiting their integration into a model-driven simulation workflow.

NoSQL databases, such as MongoDB, offer flexible, schema-less storage using JSON-like documents [66], which is beneficial for evolving or heterogeneous data structures in early-stage prototypes. PostgreSQL also supports document storage through its `JSONB` data type [67], providing a hybrid approach. However, both systems require additional tools or manual enforcement to manage schema constraints [66], and they lack native support for generating structured schemas directly from input models.

MariaDB was selected for this simulation pipeline because of its structured nature, support for standard SQL [62], and direct compatibility with automated schema generation from JSON Schema [17]. Although time-series and NoSQL databases offer advantages in specific domains [65], they do not provide built-in support for model-driven deployment, which is a core requirement of the architecture developed in this work.

Feature	MariaDB	Time-Series DBs	NoSQL
Schema Flexibility	Low (Relational schema must be predefined) [62]	Low (Schemas must be manually adapted for time-series use cases) [65]	High (Supports schema-less storage for dynamic data) [66]
Time-Series Optimization	Moderate (Requires manual table design and indexing) [62]	High (Built-in compression and window functions) [65]	Low (Not optimized for time-ordered data streams) [66]
ACID Transactions	Yes (Full transactional support) [62]	Varies (Depends on database engine and configuration) [65]	No (Focuses on availability; eventual consistency model) [66]
Query Language	Full SQL support (Standard queries with joins and indexes) [62]	Partial SQL or custom query languages (e.g., Flux in InfluxDB) [64]	JSON-like queries (Flexible but non-standard) [66]
Schema Automation	Supported (Automated via JSON Schema in this thesis) [17]	Not natively supported (Manual setup required) [65]	Limited (Possible with external validation tools) [66]
Integration with Simulation Pipelines	Good (Used per-node in this thesis with JSON-based models)	Limited (Lacks direct support for automated deployment) [65]	Moderate (Used in flexible and evolving systems) [66]
Ease of Deployment	Easy (Runs well on Debian and VirtualBox) [62]	Moderate (Often requires containerization or cloud setup) [63]	Easy (Cross-platform and easy to install) [66]
Performance with High-Frequency Data	Moderate (Requires tuning for frequent inserts) [62]	High (Optimized for real-time ingestion) [65]	Depends (Indexing and sharding affect performance) [66]

Table 3: Comparison of Database Technologies

2.3.7 *Closing the Gaps in Simulation Design*

Many existing tools focus only on isolated parts of the simulation process, such as modeling, deployment, or database management, without offering an integrated solution. These tools often lack support for DSL, runtime adaptability, and realistic system behavior, especially in smart environments. The pipeline developed in this thesis addresses these limitations by automating the full transformation from abstract schemas to executable simulations, through a combination of schema parsing, node provisioning, and REST-based inter-node coordination. This enables real-time, schema-driven behavior across distributed nodes, making simulations more flexible, realistic, and easier to adapt to diverse use cases.

Category	Tool Used	Alternative	Justification
Data Format	**JSON/JSON Schema**	XML/XML Schema	Chosen for simplicity, readability, and seamless Python integration [17].
Web Framework	**Flask**	FastAPI	Preferred for its lightweight setup and compatibility with remote deployment [54].
Database	**MariaDB**	InfluxDB / TimescaleDB	Supports structured SQL and schema automation needed for model-driven simulation [62].
Virtualization	**VirtualBox**	Docker	Provides full isolation and independent network behavior across nodes [46].
Remote Control	**Paramiko + nohup**	Ansible / Fabric	Offers flexible, low-level control without additional orchestration layers [48].
Modeling	**NetworkX**	Graph-tool	Easy to use, Python-native, and well-suited for small-to-medium network topologies [40].

Table 4: Technologies used in this thesis and reasons for selection.

3 Overview of the Transformation Pipeline

This chapter explains how the simulation system for smart environments is built and how it works. The system is designed to simulate real-world sensor networks using VMs, structured JSON-based configuration files, and lightweight web services [34, 37]. It allows researchers or developers to define their own sensor types, data formats, and communication rules, and then automatically generate and simulate a complete environment based on those inputs.

Before the pipeline can be executed, certain prerequisites must be met. These include installing the necessary software (such as VirtualBox, Python, and MariaDB), preparing a base VM image, and creating the configuration and schema files that describe the simulation environment. These setup steps are outlined in the next section.

The transformation pipeline is implemented in a modular fashion. Each module handles a specific part of the simulation workflow, and together they support a seamless process from configuration to execution. The main modules include:

- **Input Configuration:** This module uses structured JSON files to describe the environment. These files include nodes (such as sensors or rooms), their data fields (like temperature, motion, or CO_2), and the connections between them. Each node follows a schema that defines what kind of data it can generate [21].
- **Schema-Driven Data Generation:** Based on the node schemas, random but realistic data is generated for each node. For example, a node with a temperature field will get a number between the minimum and maximum values defined in its schema.
- **Graph Construction and Visualization:** The system reads the JSON configuration and builds a directed graph that shows which nodes send information to which others [40]. It then saves a visual image of the network layout for reference.
- **VM Provisioning:** For each node in the configuration, the system creates a separate VM using Oracle VirtualBox [46]. These VMs act like individual sensor nodes. They are connected through a NAT network, and each gets a unique SSH port.
- **Flask Microservice Deployment:** Each VM runs a Flask server [54] that listens for sensor data and edge messages. These servers store data in local MariaDB databases [62]. The Flask app can also respond to queries and simulate network behavior.
- **Simulation Execution:** Once the environment is running, the pipeline sends all the sensor data and edge information to the correct VMs. After the setup, users can perform SQL queries on each node, and the system measures the latency and response size for each query.
- **Monitoring and Evaluation:** The pipeline shows basic performance results, such as total simulation time and average response latency between nodes. It also prints CPU and RAM usage to help analyze system behavior.

This modular structure makes the system flexible and extensible. New sensor types can be added by defining new schema files, and different simulation scenarios can be tested by adjusting the configuration files.

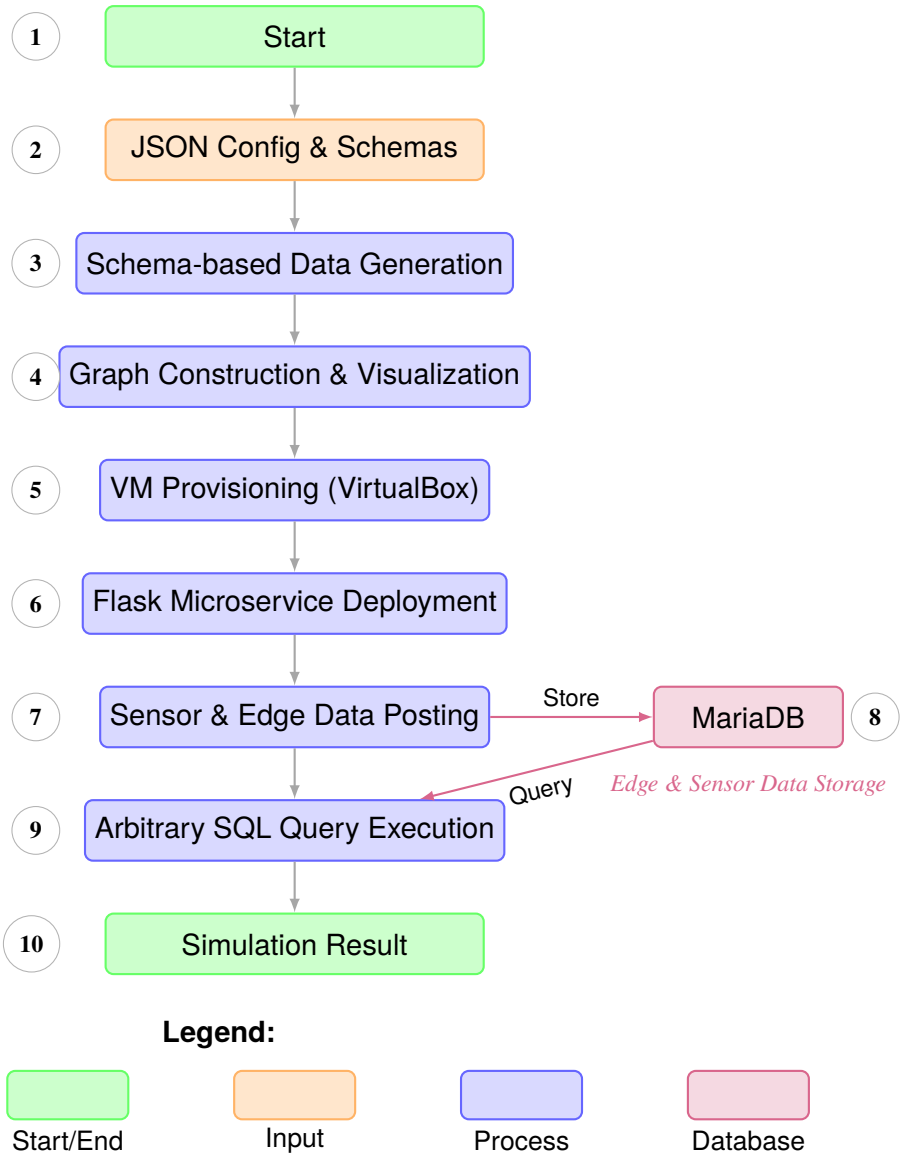


Figure 2: Simulation Transformation Pipeline

The next section outlines the setup steps required to prepare the environment. After that, each module is described in more detail with examples from the codebase and actual input files used in this thesis.

3.1 Prerequisites for Running the Pipeline

Before running the transformation pipeline, certain software dependencies and manual setup steps are required to ensure proper execution of the simulation environment. These prerequisites include:

- **Host Machine Requirements:** The system is designed to run on a Windows host with at least 8 GB of RAM and 50 GB of free disk space to accommodate multiple VM instances, simulation logs, and temporary files. Windows is required because the pipeline relies on Oracle VirtualBox's command-line interface `VBoxManage.exe`, which is referenced using absolute Windows file paths in the orchestration script.
- **VirtualBox Installation:** Oracle VirtualBox must be installed on the host system [46]. The `VBoxManage` command-line utility should be available in the system path.

- **Base Virtual Machine Image:** A preconfigured lightweight Debian-based VM image must be created manually. This image should include:
  - Python 3 and essential libraries (e.g., `Flask`, `psutil`)
  - SSH enabled with a known user/password (e.g., `rp:123`)
  - The simulation server script (e.g., `flask.py`) placed in a known location
- **JSON Configuration Files:** Users must provide a JSON file describing the simulation topology, including node definitions, edge relationships, and schema file paths.
- **Schema Files:** Each node type should have a corresponding JSON Schema file [17] defining its data structure, types, and optional constraints.
- **Python Environment:** The orchestrator script (e.g., `main.py`) should be executed in a Python 3 environment with the following packages installed: `paramiko`, `networkx`, `matplotlib`, `requests`, and `psutil`.
- **Networking Configuration:** Users should ensure that VirtualBox's NAT networking is correctly configured. Each VM must be assigned a unique SSH port via `VBoxManage` to enable independent access.

These steps only need to be completed once. After the base image and configuration files are prepared, the pipeline can be reused for different simulation scenarios by adjusting the input JSON files.

3.2 Input Configuration and Structured Specification

The simulation system begins with a structured configuration that describes the components of a smart environment and their relationships. These inputs are defined using JSON files and JSON Schema documents [58, 21]. Together, they describe the types of nodes in the system, the data each node can generate, and how nodes communicate with each other.

This section introduces the key input formats used in the system and shows how these configurations are parsed and transformed into an executable network.

3.2.1 Graph-based Topology

The topology of the smart environment is defined using JSON configuration files [40]. Each of these files contains two main sections: a `nodes` dictionary and an `edges` list.

The `nodes` section describes each component in the system. Each node has a name, a type (such as "highway", "train_detector", or "kitchen"), and a link to a JSON Schema file that defines the data structure for that node [21].

The `edges` section defines communication links between nodes. Each edge includes a source, a target, a type (such as "emergency" or "train_alert"), and the fields of data that are transmitted [7].

Below is a sample from `network_config.json`:

```
"Node1": {
  "type": "highway",
  "schema_path": "D:\\New folder (2)\\type1.json"
```

```
},
...
{ "source": "Node1", "target": "Node2", "type": "emergency", "data_fields":
    ["emergency_cars"] }
```

These configuration files are loaded by the Python function `prepare_nodes_and_graph` in the `main.py` script:

```
config = json.load(open(config_path))
node_data = {}
for name, meta in config["nodes"].items():
    schema = json.load(open(meta["schema_path"]))
    sim = generate_data_from_schema(schema)
    sim.update({"type": meta["type"], "id": name})
    node_data[name] = sim
```

This code loads the configuration and generates a dictionary of node data based on the associated schemas.

3.2.2 Node Definitions and Schemas

Each node in the system is associated with a JSON Schema that defines the fields it supports, the data types, and optional constraints such as minimum or maximum values [21]. These schemas are stored in separate files, such as:

- `type1.json`: Highway node schema
- `type2.json`: Train detector node schema
- `type3.json`: Small crossroad node schema
- `kitchen_schema.json`, `kidsroom_schema.json`, etc.: Smart home node schemas

For example, the schema in `type1.json` defines the structure of a highway node:

```
"emergency_cars": { "type": "boolean" },
"car_counter_north": { "type": "integer", "minimum": 0 },
"turn_left": { "type": "boolean" },
...
"timestamp": { "type": "string", "pattern": "^\\d{4}-\\d{2}-\\d{2} \\d{2}:
    \\d{2}:\\d{2}$" }
```

The function `generate_data_from_schema(schema)` automatically fills in values for each field based on the type and constraints:

```
def generate_data_from_schema(schema):
    data = {}
    for key, info in schema.get("properties", {}).items():
        typ = info.get("type")
        if typ == "boolean":
            data[key] = random.choice([True, False])
        elif typ == "integer":
            data[key] = random.randint(info.get("minimum", 0), info.get("maximum", 1
                00))
```

```
        elif typ == "number":
            data[key] = round(random.uniform(info.get("minimum", 0), info.get("
                maximum", 1000)), 2)
        elif typ == "string":
            data[key] = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S") if key
                 == "timestamp" else "example"
        else:
            data[key] = None
    return data
```

3.2.3 *Edge Semantics and Data Fields*

Edges in the graph define how nodes communicate. Each edge includes a direction (from source to target), a label that identifies the type of communication, and a list of data fields that will be transferred. These fields must be present in the source node's schema [21]. This structured format allows the system to simulate different scenarios by simply changing the input files, without modifying the core logic of the code.

For example, an edge may carry a list of fields like this:

```
{
  "source": "Kitchen",
  "target": "LivingRoom",
  "edge_type": "status_update",
  "data_fields": ["temperature", "gas_leak", "light_level"]
}
```

The following code shows how such edge messages are constructed and sent in the simulation:

```
payload = {
  "source": nodes[src]["id"],
  "target": nodes[tgt]["id"],
  "edge_type": edge.get("edge_type") or edge.get("type", "default_edge"),
  "data_fields": {f: nodes[src].get(f) for f in edge.get("data_fields", [])}
}
```

3.3 *Schema-driven Data Generation*

One of the key features of this simulation pipeline is that sensor data is not hardcoded or manually written. Instead, the data for each node is automatically generated based on its schema [21]. This method allows the system to stay flexible and reusable, since any new sensor type or device can be added by simply writing a new schema file.

3.3.1 *The Role of `generate_data_from_schema`*

The function `generate_data_from_schema` is responsible for creating realistic example data for each node, based on the rules defined in its JSON Schema [21]. It reads the list of properties in the schema and checks the type of each field. Then, for each property, it produces a random value that respects the allowed range or data type.

Here is the core part of the function:

```python
def generate_data_from_schema(schema):
    data = {}
    for key, info in schema.get("properties", {}).items():
        typ = info.get("type")
        if typ == "boolean":
            data[key] = random.choice([True, False])
        elif typ == "integer":
            data[key] = random.randint(info.get("minimum", 0), info.get("maximum", 1
                00))
        elif typ == "number":
            data[key] = round(random.uniform(info.get("minimum", 0), info.get("
                maximum", 1000)), 2)
        elif typ == "string":
            data[key] = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S") if key
                 == "timestamp" else "example"
        else:
            data[key] = None
    return data
```

This function uses built-in Python tools like `random.randint`, `random.uniform`, and `datetime` [68] to generate values. The result is a dictionary that mimics real sensor data.

3.3.2 Respecting Constraints from Schema Files

Each schema file defines rules for its fields. For example, some values must be within a certain range, such as a temperature between 0 and 100 degrees, or a CO_2 level between 300 and 800 ppm. Other fields are booleans that can be either `true` or `false`, like `motion_detected`, or strings like timestamps.

Here are a few sample definitions taken from our schemas:

- `"temperature": {"type": "number", "minimum": 0, "maximum": 100}`
- `"emergency_cars": {"type": "boolean"}`
- `"timestamp": {"type": "string", "pattern": "..."}`

The data generation function reads these constraints and makes sure the random values it creates follow the schema.

When this function is applied to a node, it creates an output like this (based on the kitchen schema):

```
{
  "id": "Kitchen",
  "ip": "192.168.56.101",
  "temperature": 72.4,
  "humidity": 53.8,
  "gas_leak": false,
  "motion_detected": true,
  "light_level": 682,
  "timestamp": "2025-05-18 10:32:55",
  "type": "kitchen"
}
```

This output is then sent to the Flask server running on the VM representing that node. Each node receives its own data, which reflects its specific role in the simulation.

The generated data is not only realistic, but it also helps simulate different environmental or emergency scenarios by changing the random values. This is especially useful for evaluating how the system behaves in different conditions without writing separate code for each case.

3.4 Graph Transformation and Visualization

After loading the configuration and generating the node data, the system builds a graph to represent the environment. This graph shows how different components (nodes) are connected through communication links (edges). Creating this graph is an important step, as it makes the structure of the system easy to understand and helps detect configuration issues before running the simulation.

3.4.1 Directed Graph Construction using NetworkX

The graph is built using the `NetworkX` library in Python, which is a powerful tool for creating and analyzing graphs [40]. The system uses a directed graph (`DiGraph`) because each connection has a clear direction, from a source node to a target node [40].

The code to build the graph is included in the function `draw_networkx_graph(graph)` in the `main.py` script:

```
def draw_networkx_graph(graph):
    G = nx.DiGraph()
    for node in graph["nodes"]:
        G.add_node(node["id"], **node)
    for edge in graph["edges"]:
        et = edge.get("edge_type") or edge.get("type")
        G.add_edge(edge["source"], edge["target"], type=et)
```

This function first creates an empty directed graph called `G`. It then adds all nodes with their attributes, and adds edges with the type of connection (for example, `emergency`, `status_update`, or `CO2`). This structure reflects exactly what is written in the JSON configuration files.

3.4.2 Visualization and Export (`network_graph.png`)

Once the graph is created, it is visualized using the built-in drawing functions of NetworkX and Matplotlib [69] . This helps users and researchers see the shape of the network at a glance.

The following code is used to draw the graph and save it as an image:

```
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray')
    labels = {(u, v): d['type'] for u, v, d in G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.savefig("network_graph.png")
    plt.close()
    print("[INFO] Network graph saved to: network_graph.png")
```

Here is what each part of this function does:

- `nx.spring_layout(G)` calculates positions for all nodes to make the graph look balanced and readable.

- `nx.draw(...)` draws the nodes and edges with simple styling.
- `nx.draw_networkx_edge_labels(...)` adds labels to each edge to show the type of communication.
- `plt.savefig(...)` saves the graph as a PNG file so it can be viewed later or added to documents.

The generated file `network_graph.png` provides a visual overview of the system's structure. It is especially useful when the number of nodes or edges becomes large, making the system more complex to understand just by reading the configuration files.

This visualization step completes the preparation phase of the pipeline. Now that we have valid data and a verified structure, we can move on to launching the VMs and starting the simulation services.

3.5 Virtual Environment Setup and Deployment

Once the graph is built and node data is ready, the next step is to bring the simulation to life. For this, each node in the system is assigned its own VM, creating a distributed environment where each VM behaves like an independent smart device. This section explains how the VMs are created, configured, and prepared for simulation using Oracle VirtualBox.

3.5.1 VM Cloning and Network Configuration

To create the simulation environment, a base VM image is used as a template. This image is cloned for each node. Each clone is registered with VirtualBox and started in headless mode (without a graphical interface), which helps conserve system resources [46]. The cloning itself is handled by the `clone_vm()` function in the `main.py` script:

```
def clone_vm(base, name, port):
    subprocess.run([VBOXMANAGE, "clonevm", base, "--name", name, "--register"],
        check=True)
    subprocess.run([VBOXMANAGE, "modifyvm", name, "--natpf1", f"guestssh,tcp,,{port}
        ,,22"], check=True)
    subprocess.run([VBOXMANAGE, "startvm", name, "--type", "headless"], check=True)
    print(f"Cloned {name}, SSH on port {port}")
```

This function clones a VM from the base image, assigns it a name like `Kitchen_VM`, registers it with VirtualBox, and sets up SSH port forwarding. It then starts the VM in headless mode.

3.5.2 Parallel Deployment Optimization

To support simulations with many nodes, the system uses parallel deployment to speed up the process of cloning and starting VMs. Instead of provisioning VMs one at a time, the system uses Python's `ThreadPoolExecutor` [68] to start them in parallel. This reduces the total setup time, especially when dealing with large graphs or multiple simulation scenarios.

The code snippet below shows how this is done:

```
def deploy_all_vms(config):
    with ThreadPoolExecutor(max_workers=6) as executor:
        futures = {executor.submit(deploy_vm, node_id, node_cfg): node_id
                   for node_id, node_cfg in config['nodes'].items()}
        for future in as_completed(futures):
```

```
            node_id = futures[future]
            result = future.result()
            print(f"Node {node_id}: {result}")
```

Each VM is started in its own thread, and the number of parallel workers can be adjusted based on the host machine's capabilities. This makes the pipeline more efficient and scalable, allowing it to simulate larger networks without long startup delays.

3.5.3 Port Forwarding and Host Mapping

To allow the orchestrator to connect to each VM over SSH, port forwarding is used. For example, the first VM might forward host port 2221 to guest port 22, the second VM to 2222, and so on. This setup allows each VM to be uniquely addressable via a different port on the host, even though they share the same NAT interface [50].

The following command configures port forwarding for a VM:

```
VBoxManage modifyvm VM_NAME --natpf1 "guestssh,tcp,,2221,,22"
```

This command tells VirtualBox to forward traffic from port 2221 on the host to port 22 on the guest, enabling SSH access to that VM.

3.5.4 IP Discovery and Readiness Checking

After the VMs are started, the orchestrator discovers each VM's IP address and checks if it is ready to receive commands. Since VirtualBox assigns dynamic NAT IPs internally, the orchestrator repeatedly queries the VM until a valid IP is returned.

Here is the code used to find a usable IP:

```
ip = None
retries = 10
while retries > 0 and not ip:
    out = run_ssh_command(port, "hostname -I").split()
    ip = next((x for x in out if x.startswith("192.168.56.")), None)
    if not ip:
        print(f"[WAIT] IP not ready for {name}. Retrying...")
        time.sleep(5)
        retries -= 1
```

This loop ensures that the VM has fully booted and joined the internal VirtualBox network before proceeding.

Once a valid IP is found, the orchestrator confirms that the VM is accessible via SSH using the `wait_for_ssh(port)` function:

```
def wait_for_ssh(port, timeout=200):
    start = time.time()
    while time.time() - start < timeout:
        if run_ssh_command(port, 'echo ok') == 'ok':
            return True
        time.sleep(5)
```

If the VM responds with "ok," it is marked as ready. At this point, its IP and port are saved in a configuration dictionary for later use:

```
cfgs[name] = {'host': ip, 'ssh_port': port}
```

This mapping allows the orchestrator to connect to each node individually to deploy services, send data, and run queries.

This setup step ensures that every node has a working VM, accessible IP, and known SSH port. With all VMs initialized and reachable, the system is now ready to launch simulation services, which are described in the next section.

3.6 Microservice-based Node Simulation

Each VM in the simulation acts as a separate smart node. To make this work, a lightweight Flask server is deployed inside every VM [54]. This server simulates a local service on a real sensor node. It accepts incoming data, stores it in a local MariaDB database [62], and responds to simulation queries. The Python script `flask.py` implements this behavior.

3.6.1 Flask API Implementation

The Flask server starts on port 5000 and provides several REST endpoints for interaction. It uses an environment variable to determine the name of the local database; if no name is specified, it defaults to `default_db`:

```
app = Flask(__name__)
DB = os.getenv('DB_NAME', 'default_db')
```

Before any data is handled, the database is initialized by connecting as root and granting access to a local user:

```
def setup_db():
    root_conn = mysql.connector.connect(user='root', unix_socket='/var/run/mysqld/
        mysqld.sock')
    ...
    root_cur.execute("CREATE DATABASE IF NOT EXISTS `{DB}`")
    root_cur.execute("GRANT ALL PRIVILEGES ON `{DB}`.* TO 'sensoruser'@'%'")
```

This setup ensures that each node can store its data locally using a dedicated MariaDB database [62], accessed with a simple default user account.

3.6.2 REST Endpoints and Edge Data Ingestion

The Flask server exposes multiple HTTP endpoints, each supporting a different type of simulation task:

- `GET /` – Health check; returns “Flask running”.
- `POST /data` – Receives the node’s own sensor data.
- `POST /edge` – Accepts edge messages sent from other nodes.
- `GET /query` – Executes SQL queries on the local database.
- `GET /status` – Returns a simple `<h1>OK</h1>` for quick checks.

When a node receives a message from another through an edge, it is sent to the `/edge` endpoint. The server saves this data into a table called `edge_data`:

```
@app.route('/edge', methods=['POST'])
def recv_edge():
    d = request.get_json() or {}
    ensure_columns('edge_data', d.keys())
    ...
    sql = f"INSERT INTO `edge_data` (...) VALUES (...)"
    cur.execute(sql, vals)
```

The function `ensure_columns()` ensures that any new fields in the message are automatically added to the table schema, allowing flexible updates over time.

3.6.3 Sensor Data Posting and Table Handling

Nodes also send their own internal sensor data to the `/data` endpoint. The format of this data is defined by each node's JSON Schema. For example, a kitchen node might send fields like `temperature`, `gas_leak`, and `light_level`.

The received data is stored in a dedicated table named `sensor_data`. Each new record is also tagged with a `received_at` timestamp to record when it was submitted.

```
@app.route('/data', methods=['POST'])
def recv_data():
    d = request.get_json() or {}
    ensure_columns('sensor_data', d.keys())
    ...
    sql = f"INSERT INTO `sensor_data` (...) VALUES (...)"
    cur.execute(sql, vals)
```

3.6.4 Input and Output Examples for Sensor and Edge Payloads

To help visualize the communication between components in the simulation, this section provides concrete examples of the data formats used during runtime. All messages follow JSON structure and are exchanged via HTTP requests between the orchestrator and the Flask microservices.

Sensor Data Sent to `/data`

Each node sends a payload containing generated data based on its schema. Below is an example of the data posted from a `Kitchen` node:

```
{
  "id": "Kitchen",
  "ip": "192.168.56.101",
  "temperature": 72.4,
  "humidity": 53.8,
  "gas_leak": false,
  "motion_detected": true,
  "light_level": 682,
  "timestamp": "2025-05-18 10:32:55",
  "type": "kitchen"
}
```

Edge Message Sent to `/edge`

When an edge is defined between two nodes, a structured message is created by the orchestrator and sent to the target node's Flask server:

```
{
  "source": "Kitchen",
  "target": "LivingRoom",
  "edge_type": "status_update",
  "data_fields": {
    "temperature": 72.4,
    "gas_leak": false,
    "light_level": 682
  }
}
```

SQL Response Returned from `/query`

Each Flask server supports arbitrary SQL queries. When a query such as `SELECT * FROM edge_data;` is issued, the server returns a list of rows in JSON format:

```
{
  "status": "success",
  "rows": [
    {
      "id": 1,
      "source": "Kitchen",
      "target": "LivingRoom",
      "edge_type": "status_update",
      "temperature": "72.4",
      "gas_leak": "false",
      "received_at": "2025-05-18 10:33:01"
    }
  ]
}
```

These examples show the structure and completeness of the communication flows, making the system highly testable, observable, and traceable.

3.7 Inter-Node Communication and Query Handling

A central part of this simulation system is the ability for nodes to communicate with each other, not only by sending sensor data, but also by issuing and responding to structured queries. This behavior simulates real-world scenarios where devices in a distributed system collaborate by sharing and requesting information.

Most importantly, this simulation supports arbitrary SQL queries sent from one node to another. This means that any node in the system can send a full SQL statement (like `SELECT * FROM edge_data`) to another node, wait for the result, and receive the full response. This flexible query mechanism is a unique feature of the system and one of the most important contributions of this thesis.

3.7.1 Edge Payload Construction and Dispatching

After the VMs are fully initialized and the local Flask servers are up, the orchestrator sends both sensor data and edge data to the appropriate endpoints. This includes creating structured payloads that follow the edge definitions from the JSON configuration file.

Each edge in the JSON file defines the source and target nodes, the type of connection, and the list of data fields to be transmitted. The Python function responsible for this is `send_edges()` in `main.py`:

```
for edge in config["edges"]:
    src, tgt = edge["source"], edge["target"]
    payload = {
        "source": nodes[src]["id"],
        "target": nodes[tgt]["id"],
        "edge_type": edge.get("edge_type") or edge.get("type", "default_edge"),
        "data_fields": {f: nodes[src].get(f) for f in edge.get("data_fields", [])}
    }
```

This payload is then sent using a `curl` command over SSH from the source node to the target node's Flask service:

```
cmd = (
    f"curl -s -X POST -H 'Content-Type: application/json' "
    f"-d '{json.dumps(payload)}' "
    f"http://{cfgs[tgt]['host']}:5000/edge"
)
run_ssh_command(cfgs[src]['ssh_port'], cmd)
```

This mechanism ensures that all node-to-node communication is realistic and respects the defined graph structure.

When a node receives sensor or edge data, it stores this data in its local MariaDB database. The tables `sensor_data` and `edge_data` are created if they do not exist.

3.7.2 Arbitrary SQL Query Support

One of the most powerful and flexible feature in the system is its support for arbitrary SQL queries between nodes. This allows any node to send a full SQL command to any other node and receive the exact results of that query. This capability simulates how decentralized systems in real life often need to fetch data from distributed databases.

The user is prompted to enter a custom SQL command in the orchestrator interface:

```
q = input("Enter full SQL to run :\n").strip()
sql_enc = quote_plus(q)
```

The orchestrator then sends this SQL query from every node to every other node (excluding itself), measuring latency and displaying the response.

The query is sent using the following logic:

```
url = f"http://{cfgs[t]['host']}:5000/query?sql={sql_enc}"
raw = run_ssh_command(cfgs[s]['ssh_port'], f"curl -s \"{url}\"")
```

Each Flask server has a special route `/query` that handles this request:

```
@app.route('/query', methods=['GET'])
def query():
    sql = request.args.get('sql')
    ...
    cur.execute(sql)
    rows = cur.fetchall()
    return jsonify(status='success', rows=rows), 200
```

The result is then sent back to the source node, which prints it along with the measured latency:

```
print(f"[QUERY] {s}   {t}: returned {len(rows)} rows | latency: {latency:.2f}s")
print(json.dumps(rows, indent=2))
```

This behavior makes the simulation system highly interactive. It is not limited to fixed queries or pre-programmed responses. Users or agents can issue any type of SQL command, such as `SELECT`, `COUNT`, `WHERE`, or `JOIN`, and receive real-time results from the live distributed nodes.

This feature is essential to demonstrate the potential of the system as a testbed for edge computing, distributed query systems, and dynamic smart environments. It allows detailed evaluation of performance, data distribution, query latency, and scalability, all from the perspective of arbitrary node-to-node communication.

3.8 Monitoring and Logging

To ensure that the simulation runs smoothly and to evaluate system performance, the pipeline includes basic monitoring features. These features help track how the system behaves during execution, whether nodes are responding correctly, and how long different actions take. While the current implementation does not include a centralized dashboard or advanced analytics, it still provides valuable insights using lightweight and accessible tools.

3.8.1 Basic Performance Output (Latency, CPU/RAM)

One of the main metrics tracked by the system is query latency. During the simulation, when arbitrary SQL queries are sent from one node to another, the orchestrator records how long each query takes to complete. This gives a clear picture of the communication speed between nodes and the responsiveness of the system [68].

Latency is measured using Python's `time` module:

```
t0 = time.time()
raw = run_ssh_command(cfgs[s]['ssh_port'], f"curl -s \"{url}\"")
latency = time.time() - t0
```

This value is printed for each query along with the number of returned rows:

```
print(f"[QUERY] {s}   {t}: returned {len(rows)} rows | latency: {latency:.2f}s")
```

At the end of the simulation, the system prints a summary of the overall performance:

```
print("=" * 40)
print("PERFORMANCE SUMMARY")
print("=" * 40)
print(f"Total time: {time.time() - start_total:.2f}s")
if latencies:
```

```
    print(f"Avg latency: {mean(latencies):.2f}s")
# Measure resource usage at three key stages
cpu1 = psutil.cpu_percent()
mem1 = psutil.virtual_memory().percent
# After SSH connectivity
cpu2 = psutil.cpu_percent()
mem2 = psutil.virtual_memory().percent
# After query execution
cpu3 = psutil.cpu_percent()
mem3 = psutil.virtual_memory().percent
cpu_max = max(cpu1, cpu2, cpu3)
mem_max = max(mem1, mem2, mem3)
print(f" CPU: {cpu_max}%  RAM: {mem_max}%")
```

This gives a quick overview of system load, resource usage, and average query speed, useful for identifying performance bottlenecks or delays.

3.8.2 *Logging from SSH and Flask Output*

The system also logs messages and behaviors in two ways:

- **Flask Server Logs:** When each VM starts its Flask server, it writes logs to a file called `flask.log` inside the VM. This is done by starting the server in the background using a nohup command:

```
run_ssh_command(
    cfg['ssh_port'],
    'nohup python3 /home/rp/Desktop/flask.py > flask.log 2>&1 &'
)
```

  This command redirects both standard output and errors to the log file, so that any issues with Flask startup or API handling can be reviewed later.

- **SSH Command Logs:** The orchestrator prints the result of each SSH command sent to the VMs. This includes command output, connection errors, and return messages [49]. For example, if the Flask server fails to respond, the orchestrator shows a timeout warning:

```
print(f"Flask not up {ip}:{port}")
```

  Similarly, if a command over SSH returns an error, the orchestrator prints it:

```
if err:
    print(f"SSH stderr[{port}]: {err.strip()}")
```

These simple but effective logs make it easy to track what is happening inside the system at each step. Developers and testers can check the logs to understand failures, delays, or unexpected behaviors.

3.8.3 *Fault Detection and Recovery*

The simulation pipeline includes basic routines to check the status of each virtual node and recover from common service-level failures. During execution, the orchestrator verifies whether each node is reachable via SSH, whether the Flask server is responding [54], and whether the local MariaDB instance is accessible [62]. These checks are performed at predefined stages during setup and simulation.

If a node becomes unresponsive (for example, due to a Flask crash or stalled process) the orchestrator can reconnect via SSH and restart the Flask service using the Unix `pkill` and `nohup` utilities [49]. A simplified version of the restart routine is shown below:

```
def restart_node_service(node_id, config):
    ssh_port = config['nodes'][node_id]['ssh_port']
    subprocess.run(['ssh', '-p', str(ssh_port), 'user@localhost',
                    'pkill -f "python3 ~/app.py"'], timeout=10)
    subprocess.run(['ssh', '-p', str(ssh_port), 'user@localhost',
                    'nohup python3 ~/app.py > ~/app.log 2>&1 &'], timeout=10)
```

This function terminates any existing Flask process on the node and launches a new instance in the background. Log output is redirected to a local file for later inspection.

Although these checks are not run continuously, they help ensure that each node is active and responsive during simulation. This fault recovery mechanism supports a more reliable execution flow by allowing the orchestrator to intervene when basic availability issues occur.

3.8.4 Security Notes

While the current pipeline focuses on simulation, it intentionally avoids implementing security mechanisms to keep the system lightweight and easy to debug. However, in real-world deployments or research extensions, security must be taken into account to protect sensitive data and prevent unauthorized access [54].

- **HTTP Limitations:** All communication between nodes and the orchestrator currently uses plain HTTP without encryption. This is acceptable in a local simulation environment but would not be secure on public networks.
- **Authentication:** The system does not implement any form of authentication or API key control. In a production environment, Flask endpoints would need to validate access tokens or signatures.
- **Database Access:** The MariaDB database uses a shared user account without a password. While convenient for simulation, this configuration is not secure and could be replaced with role-based access and secure credentials.

Possible Future Improvements:

- Enable HTTPS via self-signed certificates or reverse proxy (e.g., using NGINX).
- Isolate database containers per node and restrict access with user-specific credentials.
- Monitor and log query access patterns for anomaly detection or security auditing.

These improvements would make the pipeline suitable for more realistic or sensitive research environments where data integrity and privacy are critical.

3.9 Smart Environment Scenarios

The system supports multiple simulation scenarios, each designed to reflect different types of real-world smart environments. These scenarios are defined by changing the input configuration files, node

schemas, and the relationships between components. In this thesis, two key scenarios are implemented to demonstrate the flexibility and modularity of the system: an urban traffic simulation and a smart home setup. Detailed analysis and evaluation of these scenarios are presented in the next chapter. Here, we briefly describe their structure and highlight the main differences in terms of node types, communication patterns, and data interpretation.

3.9.1 Urban Traffic Simulation

This scenario models a transportation network with highways, train detectors, and small crossroads. It contains six nodes and multiple types of edges. The node types in this setup include:

- **Highway Nodes** – report vehicle counters and pedestrian crossings.
- **Train Detector Nodes** – provide train detection signals and emergency flags.
- **Small Crossroad Nodes** – send CO_2 levels and emergency data.

The edges in this setup include emergency alerts, train signals, and CO_2 reports. These messages simulate sensor interactions across the city network. Each edge carries specific data fields relevant to its source node's schema.

3.9.2 Smart Home Setup

This scenario models a residential smart environment. It includes nodes representing rooms in a home, such as:

- **Kitchen Node** – detects gas leaks, temperature, and humidity.
- **Living Room Node** – monitors motion and light levels.
- **Master Bedroom and Kids Room Nodes** – track presence, comfort, and potential alerts like crying detection.

Each node uses its own dedicated JSON Schema, such as `kitchen_schema.json` or `kidsroom_schema.json`, to define the structure of its data. Edges in this scenario carry environmental status, presence information, and safety alerts from one room to another.

3.9.3 Key Differences Between Scenarios

The two simulation cases differ in several important ways:

- **Data Fields:** Urban nodes handle counts and emergencies (e.g., car counters, pedestrian flags), whereas smart home nodes deal with comfort and safety (e.g., light levels, gas leaks, crying detection).
- **Edge Semantics:** In the traffic setup, edges represent network-wide alerts such as train warnings or emergency routing. In the home setup, edges carry more localized messages, such as presence updates or motion detection between rooms.

These two cases demonstrate the adaptability of the pipeline to different domains. They also serve as the basis for the evaluation in the next chapter, where each scenario is analyzed in terms of performance, accuracy, and simulation behavior.

3.10 Implementation Summary and Design Insights

This chapter presented the complete implementation of the simulation pipeline used to model smart environments. The system is designed to be flexible, modular, and easy to extend, enabling the simulation of various real-world scenarios with minimal configuration effort.

The pipeline begins with structured JSON files that define the environment's topology, the properties of each node, and the communication rules. Each node is paired with a JSON Schema [17], which is used to generate realistic synthetic sensor data. A directed graph is constructed from this configuration, visualized using NetworkX [40], and saved as an image for reference.

Each node is deployed as a separate VM, acting as an independent simulation of a smart device. Inside each VM, a lightweight Flask server [54] exposes a REST API to receive data, store it in a local MariaDB database, and handle SQL queries. Nodes communicate through HTTP requests along edges defined in the configuration. Database schemas are adjusted dynamically to accommodate new fields in incoming data.

A key design choice is the use of decoupled microservices: each node runs its own isolated server and database. This models real-world distributed environments more accurately and enables simulation of complex inter-node interactions.

Another important decision is the reliance on schema-driven data generation. By using JSON Schema files, the system avoids hardcoded logic and supports domain-specific customization simply by providing new schema definitions.

Finally, the system supports arbitrary SQL queries between nodes. This feature allows users to inspect the internal state of any node and simulate rich data interactions, reflecting real-world requirements in edge computing and distributed systems.

Together, these design decisions result in a simulation platform that is both powerful and adaptable, one that supports research into smart environments, data pipelines, and distributed coordination.

3.10.1 Modularity, Extensibility, and Parameterization

The transformation pipeline is designed to be flexible and easy to reuse. To support different kinds of smart environments, it was intentionally built to be modular, extensible, and configurable through input parameters. This section explains how each of these aspects works in practice.

Modularity

The pipeline follows a modular structure, where each main task, such as reading input files, generating data, creating VMs, and simulating communication, is handled by a separate component. Each module has clear inputs and outputs, which makes the system easier to understand, change, and test.

This separation also means that parts of the system can be swapped or improved without affecting the rest. For example:

- The input configuration can be changed to model a different type of environment, like switching from a traffic network to a smart building, just by editing the JSON files and schemas.
- The database engine, currently MariaDB, could be replaced with another system like PostgreSQL, as long as it supports similar SQL features.

- The deployment process could use Docker or Kubernetes instead of VirtualBox, with only small changes to how VMs are started.

Extensibility

Although the current version already supports a variety of simulation behaviors, it can be extended easily. The Python code is written to allow new features to be added with minimal effort.

Here are a few examples of what could be added:

- New API endpoints in the Flask application to handle extra commands or query types.
- New node types, created by writing new JSON Schema files and defining their behavior.
- Extra logic in the query system, such as rules or triggers based on the schema, so nodes can respond more intelligently.

Since the system already allows flexible SQL queries and supports dynamic schema updates, it provides a strong base for more advanced research in areas like distributed reasoning, secure communication, or collaborative data sharing.

Parameterization

Another strength of the system is that many parts of the simulation can be changed through input files, without touching the main code. This makes it easy to experiment with different scenarios or run quick tests.

Parameterization is done through:

- **JSON Schemas:** Each node can have its own schema that defines its data fields, types, and value ranges.
- **Graph Configuration:** Connections and data flows are set in the edge list JSON file, so the user can change how nodes interact just by editing one file.
- **Runtime Parameters:** Things like SQL query content, test size, or node roles can be adjusted at runtime using orchestrator inputs.

For example, to test how the system reacts to faulty sensors, a user could add noise to the data generator. To test scalability, one could duplicate more nodes and expand the graph to simulate a larger environment.

These three design features (modularity, extensibility, and parameterization) make the pipeline a powerful tool for simulating and experimenting with different types of smart environments. Whether the goal is to test a traffic network, a home automation system, or an industrial setup; the same system can be reused and adapted with little effort.

4 Evaluation Methodology and Case-Based Analysis

This chapter presents the evaluation approach used to assess the transformation pipeline. The goal is not only to validate the system's functional correctness but also to examine its scalability, robustness, and adaptability to various real-world smart environment scenarios.

To ensure a well-rounded assessment, both quantitative and qualitative criteria are applied. The evaluation is structured around two distinct case studies. The first focuses on an urban traffic simulation, where the pipeline is tested in a context involving emergency-aware road segments, train detection nodes, and small crossroads equipped with environmental sensors. This scenario emphasizes distributed communication and timely inter-node coordination, offering insights into how well the system scales under higher network and data interaction loads.

The second case study explores a smart home environment, comprising interconnected rooms such as a kitchen, living room, bedroom, and children's room. Each room functions as an independent node, equipped with custom sensors defined through individual schemas. This setup allows evaluation of how effectively the pipeline supports semantic flexibility, schema diversity, and scenario-specific behaviors such as CO_2 monitoring, light control, or motion detection.

Together, these case studies demonstrate not only the correctness of the generated simulations but also the versatility and robustness of the pipeline. By evaluating the system across domains with different requirements and interaction patterns, the chapter illustrates the pipeline's potential as a reusable framework for simulating smart environments at various levels of complexity.

4.1 Evaluation Methodology and Measurement Setup

To evaluate the transformation pipeline, both quantitative and qualitative criteria were used. The goal was to assess the system's correctness, performance, scalability, robustness, and extensibility across different smart environment scenarios. All results were gathered over five repeated simulation runs for each scenario, and the outcomes remained consistent and stable across these iterations.

Correctness was treated as a binary outcome for each node: a node was considered correct if it successfully booted, responded to SSH, launched its local Flask service, processed its schema-defined data through the `/data` endpoint, and handled incoming edge messages at the `/edge` endpoint, storing them in its local MariaDB database. The entire system was considered correct if all nodes met these criteria. Failures could occur if the input configuration was invalid (e.g., referencing undefined nodes), if a schema was malformed or inconsistent with the database structure, if a VM or Flask server failed to start, or if a node mishandled unexpected input fields. These are real possibilities based on the pipeline's modular execution flow, where model definitions are interpreted dynamically. Although the pipeline reads the model and builds the system accordingly, runtime errors or mismatches can still arise if the input is structurally valid but semantically flawed, such as a missing expected field in a data message.

Performance was evaluated by measuring how quickly the pipeline transitions from model input to a fully operational simulation environment. This includes the time taken to launch VMs, start Flask services, and complete schema-based configuration. These steps are run in parallel where possible using Python's `ThreadPoolExecutor`, reducing total setup time as the number of nodes increases.

Scalability was tested by varying the number of nodes and edges in the simulation.

In addition to these measurable aspects, the system's robustness and extensibility were evaluated qualitatively. Robustness was confirmed by simulating faults, such as delayed Flask startups or manual

service termination. In both cases, the orchestrator detected the issue and restarted the service remotely using SSH and `nohup` [49], without manual intervention. Extensibility was demonstrated by adding new node types, modifying schemas, and rewiring the graph entirely through configuration files, without changing core code.

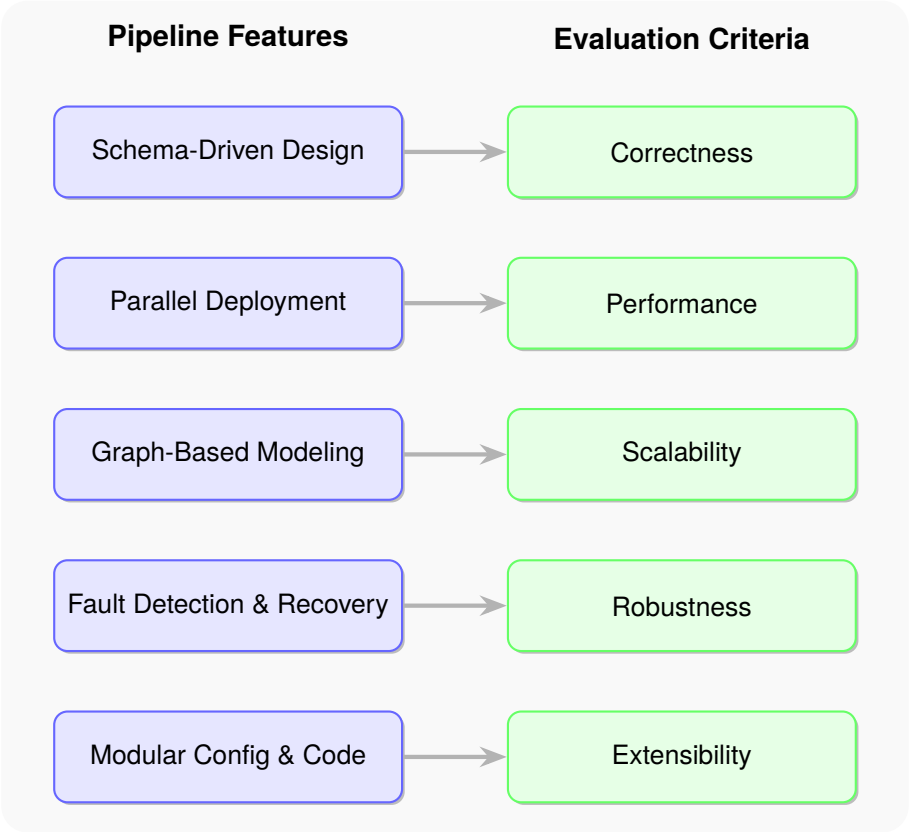


Figure 3: Mapping of pipeline features to evaluation criteria.

The following sections apply these evaluation methods to two case studies, urban traffic simulation and smart home environments, to assess how well the pipeline meets these criteria in practice. Through these experiments, the goal is to observe the system's behavior under different conditions, measure its performance, and explore its flexibility and reliability across varied smart environment scenarios.

4.2 Case Study 1: Urban Traffic Management

To evaluate the transformation pipeline in a realistic application domain, we developed a simulation of a smart urban traffic management system. The goal of this case study is to replicate the interactions between various traffic control nodes in a small city segment and assess the system's ability to handle both routine data exchange and emergency-related events.

The simulated environment includes three functional node types: highway controllers, train detectors, and small crossroads. All nodes share a set of common capabilities, including directional vehicle counting, pedestrian detection, and emergency vehicle monitoring. Each type is also extended with specialized functionality: highway nodes detect turning movements (left/right), train detector nodes sense approaching trains, and small crossroads monitor air quality using CO_2 sensors.

These nodes interact continuously by exchanging information based on predefined communication rules. Emergency alerts and train detection signals are routed only to directly connected neighbor nodes,

enabling localized responses. In contrast, CO_2 broadcasts are sent to all nodes, supporting system-wide environmental awareness. These differentiated message flows are encoded in the system's graph configuration, with edge definitions specifying directionality and data fields. The transformation pipeline interprets this model to automatically configure each node's communication behavior.

For instance, when a train is detected, the corresponding node notifies its immediate neighbors to initiate traffic-halting procedures. If a crossroad node detects elevated CO_2 levels, it broadcasts this information to all nodes, potentially influencing higher-level control mechanisms. Similarly, when an emergency vehicle is detected, the responsible node sends alerts to its adjacent nodes, enabling coordinated traffic light adjustments to prioritize a clear travel path.

Figure 4: Simulated urban layout showing node types and their positions. Visualization by Nima Farahmandnia, used with permission. The system consists of six nodes: two train detectors (bottom), two highway nodes (center), and two small crossroads (top).

Each node operates inside its own VM, running a Flask-based API for inter-node communication and a local MariaDB instance for logging and querying sensor data.

Node interactions follow three primary messaging patterns: emergency alerts, train detection signals, and CO_2 broadcasts. Each of these is represented as a distinct edge type in the system's graph configuration. For instance, the internal configuration includes entries such as:

- `type: "train_alert"` from `Node4` to `Node6`, carrying the field `train_detection`,
- `type: "emergency"` from `Node4` to `Node6`, with `emergency_cars` as the data field,
- `type: "CO2"` from `Node5` to `Node1`, transmitting `CO2` sensor readings.

These edges define not only the sender and receiver nodes but also the direction and semantic type of communication. During deployment, the pipeline automatically reads these edge definitions to configure

routing behavior, message payloads, and communication endpoints for each node. As a result, all inter-node communication is driven entirely by the graph model, without requiring hardcoded logic.

4.2.1 Scenario Graphs and Edge Topologies

To assess the flexibility and scalability of the transformation pipeline, we designed and tested three distinct simulation scenarios. Each scenario reflects a different configuration of smart urban traffic nodes with a mix of emergency logic, environmental sensors, and train detection. The input models were described using JSON files and processed by the pipeline without modifying any Python source code.

Scenario 1: Emergency and Train-Alert Logic (3 Nodes)

This baseline configuration includes three nodes:

- Node1, 2: A highway segment with sensors for emergency vehicles, turning cars, and car counters.
- Node3: A train detector that monitors rail crossings and sends alerts.

Each node sends emergency alerts to its neighbors. The train detector also triggers `train_alert` messages to prevent traffic buildup during train crossings.

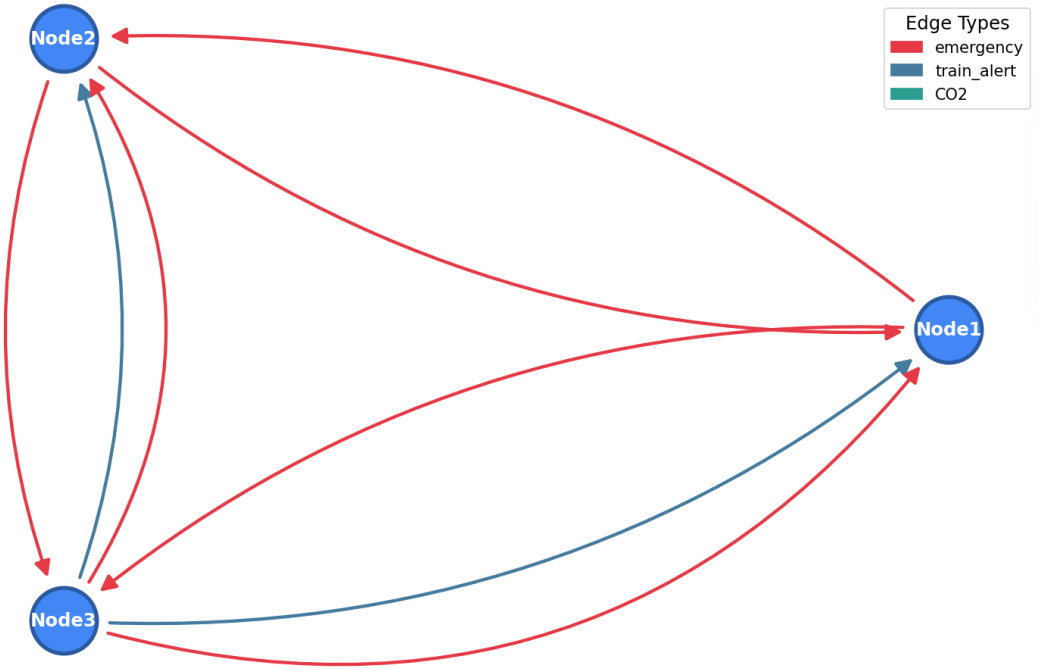


Figure 5: Scenario 1 – Visualized graph showing directional edge types between nodes.

The plot above demonstrates the interaction logic: red edges represent emergency messages, while blue edges show train alerts directed to relevant nodes. All communication is defined by node type and schema logic.

```
All original edges (with keys):
Node1 -> Node2 (key=0), type=emergency, fields=['emergency_cars']
Node1 -> Node3 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node1 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node3 (key=0), type=emergency, fields=['emergency_cars']
Node3 -> Node2 (key=0), type=train_alert, fields=['train_detection']
Node3 -> Node2 (key=1), type=emergency, fields=['emergency_cars']
Node3 -> Node1 (key=0), type=train_alert, fields=['train_detection']
Node3 -> Node1 (key=1), type=emergency, fields=['emergency_cars']
```

Figure 6: Scenario 1 – Extracted edge list printed during transformation.

Figure 6 confirms that all emergency and train alert messages are mapped correctly. Emergency edges use the `emergency_cars` field, while train alerts rely on the `train_detection` attribute defined in the train node's schema.

Scenario 2: CO_2 Broadcasting with Partial Isolation (3 Nodes)

In second setup, we again use three nodes, but with a different logic structure. Node3 in this scenario represents a small crossroad equipped solely with CO_2 sensors. Unlike the other two nodes, it is not connected to any of them through emergency or train alert edges. However, because its schema defines a CO_2 output field, it still sends environmental updates to all nodes in the network.

- Node1: Highway controller, capable of sending and receiving emergency signals.
- Node2: Train detector node, which emits targeted train alerts and emergency signals.
- Node3: Small crossroad node with CO_2 sensor, but no declared emergency or train-related behavior.

Although Node3 has no incoming or outgoing emergency or train connections, the pipeline automatically creates CO_2 edges to all other nodes because environmental broadcasts are handled as global signals in the transformation logic.

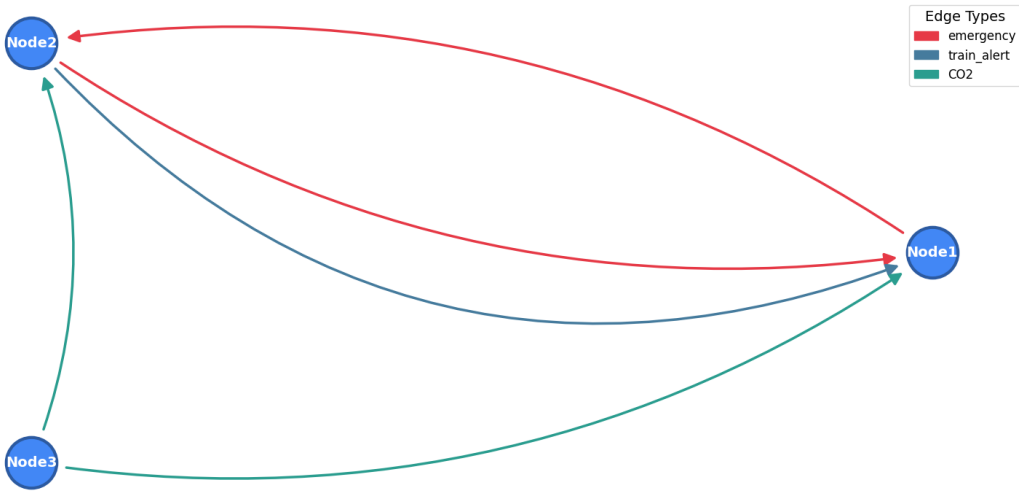


Figure 7: Scenario 2 – Visualized graph showing directional edge types between nodes.

In Figure 7, Node3 has no red or blue arrows (emergency or train alert). Only green edges flow from it, representing CO_2 data directed to both Node1 and Node2. This setup emphasizes that edge generation is entirely governed by schema fields rather than structural connectivity alone.

```
All original edges (with keys):
Node1 -> Node2 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node1 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node1 (key=1), type=train_alert, fields=['train_detection']
Node3 -> Node2 (key=0), type=CO2, fields=['CO2']
Node3 -> Node1 (key=0), type=CO2, fields=['CO2']
```

Figure 8: Scenario 2 – Extracted edge list printed during transformation.

As confirmed in the edge log shown in Figure 8, Node3 only participates in CO_2 transmissions using the field `CO2`. Neither emergency signals nor train alerts are associated with Node3. This behavior was generated entirely from schema analysis and validates the pipeline's selective message mapping.

This scenario is particularly useful to evaluate schema-based isolation: despite existing in the network, Node3 behaves independently in terms of emergency and train signaling but still contributes globally relevant environmental data to the system.

Scenario 3: Dense Urban Simulation with Full Connectivity (6 Nodes)

The final scenario expands to six nodes, two of each type (highway, train detector, small crossroad), to emulate a densely connected smart traffic network. This graph allows testing of scalability and multi-event propagation under realistic load.

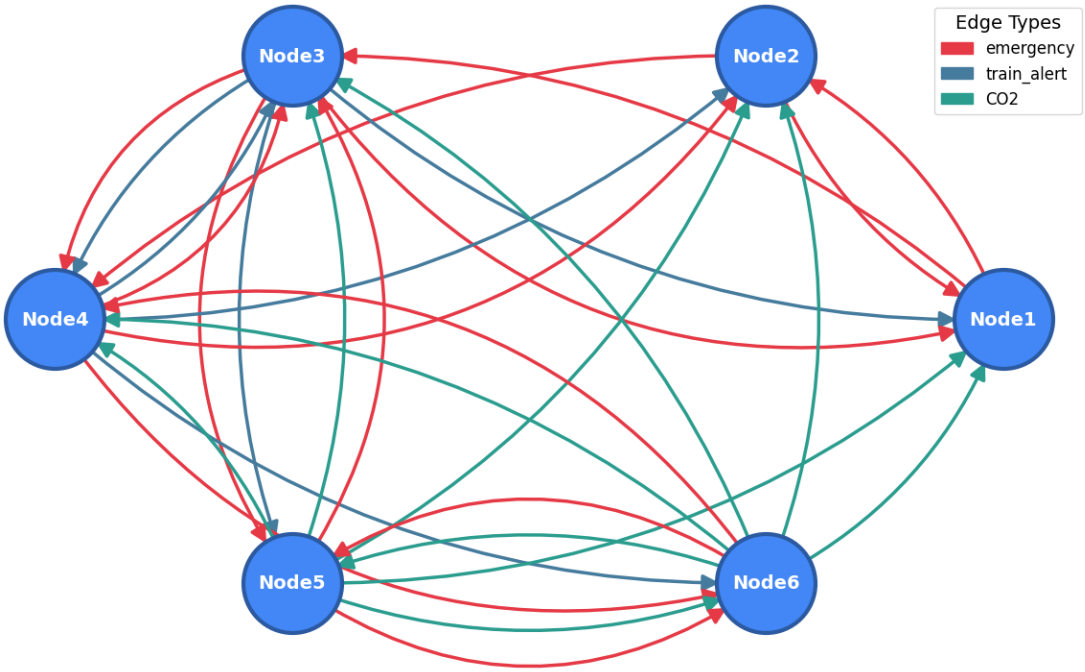


Figure 9: Scenario 3 – Visualized graph showing directional edge types between nodes.

The figure shows a high volume of messages of all types (emergency, train alerts, CO_2) crisscrossing the network. Each color-coded edge corresponds to one type of communication logic, generated directly from schema analysis.

```
All original edges (with keys):
Node1 -> Node2 (key=0), type=emergency, fields=['emergency_cars']
Node1 -> Node3 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node1 (key=0), type=emergency, fields=['emergency_cars']
Node2 -> Node4 (key=0), type=emergency, fields=['emergency_cars']
Node3 -> Node1 (key=0), type=train_alert, fields=['train_detection']
Node3 -> Node1 (key=1), type=emergency, fields=['emergency_cars']
Node3 -> Node4 (key=0), type=train_alert, fields=['train_detection']
Node3 -> Node4 (key=1), type=emergency, fields=['emergency_cars']
Node3 -> Node5 (key=0), type=train_alert, fields=['train_detection']
Node3 -> Node5 (key=1), type=emergency, fields=['emergency_cars']
Node4 -> Node2 (key=0), type=train_alert, fields=['train_detection']
Node4 -> Node2 (key=1), type=emergency, fields=['emergency_cars']
Node4 -> Node3 (key=0), type=train_alert, fields=['train_detection']
Node4 -> Node3 (key=1), type=emergency, fields=['emergency_cars']
Node4 -> Node6 (key=0), type=train_alert, fields=['train_detection']
Node4 -> Node6 (key=1), type=emergency, fields=['emergency_cars']
Node5 -> Node1 (key=0), type=CO2, fields=['CO2']
Node5 -> Node2 (key=0), type=CO2, fields=['CO2']
Node5 -> Node3 (key=0), type=CO2, fields=['CO2']
Node5 -> Node3 (key=1), type=emergency, fields=['emergency_cars']
Node5 -> Node4 (key=0), type=CO2, fields=['CO2']
Node5 -> Node6 (key=0), type=CO2, fields=['CO2']
Node5 -> Node6 (key=1), type=emergency, fields=['emergency_cars']
Node6 -> Node1 (key=0), type=CO2, fields=['CO2']
Node6 -> Node2 (key=0), type=CO2, fields=['CO2']
Node6 -> Node3 (key=0), type=CO2, fields=['CO2']
Node6 -> Node4 (key=0), type=CO2, fields=['CO2']
Node6 -> Node4 (key=1), type=emergency, fields=['emergency_cars']
Node6 -> Node5 (key=0), type=CO2, fields=['CO2']
Node6 -> Node5 (key=1), type=emergency, fields=['emergency_cars']
```

Figure 10: Scenario 3 – Extracted edge list printed during transformation.

The transformation pipeline parses each edge, determines its type from node schemas, and logs the associated field (e.g., `emergency_cars`, `CO2`, `train_detection`). As shown in Figure 10, the edges are consistently derived, covering all required interactions and confirming system correctness at larger scales.

Each of these scenarios illustrates a different challenge: maintaining behavioral correctness in a minimal setup, handling selective versus broadcast communication, and scaling to complex urban configurations.

4.2.2 Query and Behavior Verification

To verify node behavior and data consistency during simulation, each Flask server includes a `/query` endpoint that executes SQL statements against the local MariaDB instance. Queries are sent from a source node to a target node, where they are executed locally, and the results are then returned back to the source. This mechanism enables runtime inspection of both sensor data and messages received from other nodes, supporting decentralized and on-demand behavior validation.

In the earlier version of the system, this endpoint was hardcoded to execute a specific SQL query that checked for emergency vehicle detection. The logic returned a simplified traffic status ("Green light" or "Red light") based on whether any matching sensor data was found. The predefined query was:

```
SELECT COUNT(*) FROM sensor_data
WHERE JSON_EXTRACT(data, '$.emergency_cars') = 'true';
```

If the count was greater than zero, the system returned "Green light" to simulate traffic flow priority for emergency vehicles; otherwise, it returned "Red light." This logic provided a basic demonstration of reactive behavior but was limited to a fixed use case.

In the current implementation, the `/query` endpoint supports dynamic SQL execution. Queries are passed via the `sql` parameter and can be written manually or programmatically during simulation. This design supports more flexible, context-aware data analysis without requiring changes to the server code.

Two representative SQL queries were used in the urban traffic scenario to test dynamic behavior:

- **Query 1: Aggregated vehicle counts by direction.**
  This query sums directional vehicle counts from the `sensor_data` table:

```sql
SELECT SUM(car_counter_north) AS total_north,
       SUM(car_counter_south) AS total_south,
       SUM(car_counter_east) AS total_east,
       SUM(car_counter_west) AS total_west,
       (SUM(car_counter_north) + SUM(car_counter_south) +
        SUM(car_counter_east) + SUM(car_counter_west)) AS total_all_directions
FROM sensor_data;
```

- **Query 2: Emergency status from received messages.**
  This query checks whether any emergency vehicle messages were received by the node via the `edge_data` table:

```sql
SELECT CASE WHEN EXISTS (
    SELECT 1 FROM edge_data
    WHERE JSON_EXTRACT(data_fields, '$.emergency_cars') = true
) THEN 'emergency' ELSE 'normal' END AS status;
```

Query results were printed in the orchestrator terminal during runtime, confirming that the system correctly processed and responded to live, schema-driven data. This shift from predefined logic to fully dynamic querying illustrates the pipeline's flexibility in supporting diverse simulation scenarios.

Scenario 1: Baseline 3-Node Setup

In this first scenario, three nodes were connected using a minimal configuration involving `emergency` and `train_alert` edges. To verify node behavior, we evaluated both the earlier version of the system, which used a predefined query, and the current version, which supports dynamic query execution.

As shown in Figure 11, in the earlier version all nodes responded with `"Red light"`, confirming that no emergency vehicles were detected in the dataset. Response latencies ranged between 2.87 and 2.94 seconds, demonstrating reliable system responsiveness even under the hardcoded logic.

```
[QUERY] Node1→Node2: Red light | latency: 2.94 sec
[QUERY] Node1→Node3: Red light | latency: 2.93 sec
[QUERY] Node2→Node1: Red light | latency: 2.88 sec
[QUERY] Node2→Node3: Red light | latency: 2.88 sec
[QUERY] Node3→Node1: Red light | latency: 2.87 sec
[QUERY] Node3→Node2: Red light | latency: 2.90 sec
```

Figure 11: Scenario 1 – Predefined emergency query results.

In the current version, behavior was verified using dynamic SQL queries issued externally to each node during runtime. Two representative queries were used:

- One to check for emergency alerts based on data in the `edge_data` table (Query 2).

- One to calculate the total number of detected vehicles from all directions using the `sensor_data` table (Query 1).

As shown in Figure 12, all emergency query responses returned `"normal"`, indicating that no node had received emergency messages from its neighbors. This confirms that the edge message logic functioned as expected in the absence of emergency events.

```
SELECT CASE WHEN EXISTS (SELECT 1 FROM edge_data WHERE JSON_EXTRACT(data_fields, '$.emergency_cars') = true) THEN 'emergency' ELSE 'normal' END AS status;
[QUERY] Node1→Node2: returned 1 rows | latency: 3.11s
[
  {
    "status": "normal"
  }
]
[QUERY] Node1→Node3: returned 1 rows | latency: 2.85s
[
  {
    "status": "normal"
  }
]
[QUERY] Node2→Node1: returned 1 rows | latency: 2.83s
[
  {
    "status": "normal"
  }
]
[QUERY] Node2→Node3: returned 1 rows | latency: 2.81s
[
  {
    "status": "normal"
  }
]
[QUERY] Node3→Node1: returned 1 rows | latency: 2.86s
[
  {
    "status": "normal"
  }
]
[QUERY] Node3→Node2: returned 1 rows | latency: 2.73s
[
  {
    "status": "normal"
  }
]
```

Figure 12: Scenario 1 – Emergency query responses.

The second query computed total vehicle counts based on each node's directional sensor data. As shown in Figure 13, results varied across nodes, reflecting differences in local traffic conditions and sensor values.

```
SELECT SUM(car_counter_north) AS total_north, SUM(car_counter_south) AS total_south, SUM(car_counter_east) AS total_east, SUM(car_counter_west) AS total_wes
t, (SUM(car_counter_north)+SUM(car_counter_south)+SUM(car_counter_east)+SUM(car_counter_west)) AS total_all_directions FROM sensor_data;
[QUERY] Node1→Node2: returned 1 rows | latency: 2.79s
[
  {
    "total_all_directions": 71.0,
    "total_east": 8.0,
    "total_north": 4.0,
    "total_south": 15.0,
    "total_west": 44.0
  }
]
[QUERY] Node1→Node3: returned 1 rows | latency: 3.00s
[
  {
    "total_all_directions": 198.0,
    "total_east": 41.0,
    "total_north": 18.0,
    "total_south": 89.0,
    "total_west": 50.0
  }
]
[QUERY] Node2→Node1: returned 1 rows | latency: 3.21s
[
  {
    "total_all_directions": 174.0,
    "total_east": 51.0,
    "total_north": 31.0,
    "total_south": 5.0,
    "total_west": 87.0
  }
]
[QUERY] Node2→Node3: returned 1 rows | latency: 2.95s
[
  {
    "total_all_directions": 198.0,
    "total_east": 41.0,
    "total_north": 18.0,
    "total_south": 89.0,
    "total_west": 50.0
  }
```

Figure 13: Scenario 1 – Vehicle count totals returned from `sensor_data`.

All queries were executed with low latency (ranging from 2.73s to 3.21s), confirming that the system maintained reliable performance under minimal load. This scenario demonstrates that the system:

- Correctly responds to both local sensor readings and received messages.
- Supports flexible, runtime query inspection through dynamic SQL.
- Maintains consistent response times across the simulated environment.

Scenario 2: CO_2-Broadcasting Crossroad Node

In this scenario, the network was extended to include a crossroad node (Node3) equipped with a CO_2 sensor. This configuration was used to evaluate whether emergency-related behavior would continue to function correctly alongside unrelated sensor logic.

As in the previous scenario, a predefined query was used first to check for emergency vehicle detections. Each node evaluated its local `sensor_data` table and returned either `"Red light"` or `"Green light"` depending on the result. As shown in Figure 14, all nodes returned `"Red light"`, confirming that no emergency data was present in the local records. Latencies ranged between 2.78 and 2.95 seconds.

```
[QUERY] Node1→Node2: Red light | latency: 2.87 sec
[QUERY] Node1→Node3: Red light | latency: 2.81 sec
[QUERY] Node2→Node1: Red light | latency: 2.82 sec
[QUERY] Node2→Node3: Red light | latency: 2.95 sec
[QUERY] Node3→Node1: Red light | latency: 2.78 sec
[QUERY] Node3→Node2: Red light | latency: 2.93 sec
```

Figure 14: Scenario 2 – Predefined emergency query results.

Next, a dynamic emergency status query was issued, checking for messages containing the `emergency_cars` field in each node's `edge_data`. As shown in Figure 15, all valid responses returned `"emergency"`, confirming that emergency alerts had been received. However, two queries directed to Node3 resulted in SQL errors due to the absence of the `data_fields` column. This occurred because Node3 had not received any messages containing this field, and thus MariaDB never created the column during table evolution.

The following error was returned:

```
ERROR 1054 (42S22): Unknown column 'data_fields' in 'where clause'
```

This confirms the dynamic nature of schema evolution in the system: columns are only created if corresponding data is received. Importantly, these errors affected only the specific queries sent to Node3, while other nodes continued to respond normally and without interruption.

```
Enter full SQL to run :
SELECT CASE WHEN EXISTS (SELECT 1 FROM edge_data WHERE JSON_EXTRACT(data_fields, '$.emergency_cars') = true) THEN 'emergency' ELSE 'normal' END AS status;
[QUERY] Node1→Node2: returned 1 rows | latency: 2.84s
[
  {
    "status": "emergency"
  }
]
[QUERY] Node1→Node3: ERROR 1054 (42S22): Unknown column 'data_fields' in 'where clause' | latency: 2.84s
[QUERY] Node2→Node1: returned 1 rows | latency: 3.20s
[
  {
    "status": "emergency"
  }
]
[QUERY] Node2→Node3: ERROR 1054 (42S22): Unknown column 'data_fields' in 'where clause' | latency: 2.80s
[QUERY] Node3→Node1: returned 1 rows | latency: 2.74s
[
  {
    "status": "emergency"
  }
]
[QUERY] Node3→Node2: returned 1 rows | latency: 2.74s
[
  {
    "status": "emergency"
  }
]
------------------------------------
```

Figure 15: Scenario 2 – Emergency query responses.

Finally, the dynamic vehicle count query was executed. This query aggregated directional traffic counts from the `sensor_data` table on each node. All queries completed successfully and returned

totals that accurately reflected simulated local traffic conditions. Latencies were slightly improved in this step, ranging from 2.48 to 2.88 seconds.

```
Enter full SQL to run :
SELECT SUM(car_counter_north) AS total_north, SUM(car_counter_south) AS total_south, SUM(car_counter_east) AS total_east, SUM(car_counter_west) AS total_wes
t, (SUM(car_counter_north)+SUM(car_counter_south)+SUM(car_counter_east)+SUM(car_counter_west)) AS total_all_directions FROM sensor_data;
[QUERY] Node1→Node2: returned 1 rows | latency: 2.88s
[
  {
    "total_all_directions": 261.0,
    "total_east": 24.0,
    "total_north": 76.0,
    "total_south": 71.0,
    "total_west": 90.0
  }
]
[QUERY] Node1→Node3: returned 1 rows | latency: 2.76s
[
  {
    "total_all_directions": 165.0,
    "total_east": 52.0,
    "total_north": 21.0,
    "total_south": 23.0,
    "total_west": 69.0
  }
]
[QUERY] Node2→Node1: returned 1 rows | latency: 2.64s
[
  {
    "total_all_directions": 181.0,
    "total_east": 26.0,
    "total_north": 89.0,
    "total_south": 66.0,
    "total_west": 0.0
  }
]
[QUERY] Node2→Node3: returned 1 rows | latency: 2.48s
[
  {
    "total_all_directions": 165.0,
    "total_east": 52.0,
    "total_north": 21.0,
    "total_south": 23.0,
    "total_west": 69.0
```

Figure 16: Scenario 2 – Vehicle count totals returned from `sensor_data`.

Overall, this scenario demonstrates that:

- Schema evolution is node-specific and depends entirely on received message structure.
- Query failures are handled independently and do not affect other nodes' performance or accuracy.

Scenario 3: Six Nodes with Mixed Sensor States

In the third and most comprehensive scenario, the simulation was scaled up to include six nodes representing a mix of highways, train detectors, and crossroad sensors. This setup was designed to test the pipeline's ability to maintain accurate messaging, local state tracking, and query performance under a more complex configuration.

As in previous scenarios, emergency status was first evaluated using a predefined query. Each node analyzed its own `sensor_data` and returned either `"Red light"` or `"Green light"` depending on whether an emergency vehicle had been detected. As shown in Figure 17, several nodes responded with `"Green light"`, confirming that local emergency conditions were present in those environments, while others returned `"Red light"`.

```
[QUERY] Node1→Node2: Red light | latency: 2.91 sec
[QUERY] Node1→Node3: Green light | latency: 2.85 sec
[QUERY] Node1→Node4: Green light | latency: 2.85 sec
[QUERY] Node1→Node5: Green light | latency: 2.86 sec
[QUERY] Node1→Node6: Red light | latency: 2.85 sec
[QUERY] Node2→Node1: Red light | latency: 2.84 sec
[QUERY] Node2→Node3: Green light | latency: 2.85 sec
[QUERY] Node2→Node4: Green light | latency: 2.86 sec
[QUERY] Node2→Node5: Green light | latency: 2.86 sec
[QUERY] Node2→Node6: Red light | latency: 2.87 sec
[QUERY] Node3→Node1: Red light | latency: 2.87 sec
[QUERY] Node3→Node2: Red light | latency: 2.91 sec
[QUERY] Node3→Node4: Green light | latency: 2.94 sec
[QUERY] Node3→Node5: Green light | latency: 2.82 sec
[QUERY] Node3→Node6: Red light | latency: 2.86 sec
[QUERY] Node4→Node1: Red light | latency: 2.80 sec
[QUERY] Node4→Node2: Red light | latency: 2.87 sec
[QUERY] Node4→Node3: Green light | latency: 2.84 sec
[QUERY] Node4→Node5: Green light | latency: 2.90 sec
[QUERY] Node4→Node6: Red light | latency: 2.86 sec
[QUERY] Node5→Node1: Red light | latency: 2.93 sec
[QUERY] Node5→Node2: Red light | latency: 2.95 sec
[QUERY] Node5→Node3: Green light | latency: 2.81 sec
[QUERY] Node5→Node4: Green light | latency: 2.83 sec
[QUERY] Node5→Node6: Red light | latency: 2.88 sec
[QUERY] Node6→Node1: Red light | latency: 3.19 sec
[QUERY] Node6→Node2: Red light | latency: 2.86 sec
[QUERY] Node6→Node3: Green light | latency: 2.83 sec
[QUERY] Node6→Node4: Green light | latency: 2.82 sec
[QUERY] Node6→Node5: Green light | latency: 2.85 sec
```

Figure 17: Scenario 3 – Predefined emergency query results.

A dynamic emergency status query was then used to assess whether nodes had received emergency messages from others via their `edge_data`. Figure 18 shows a representative subset of the results. All visible responses returned `"normal"`, indicating no received emergency messages were recorded in the sampled nodes. While not all node responses are shown in the output, the consistency of the visible results suggests correct schema handling and absence of emergency propagation in this specific run.

```
SELECT CASE WHEN EXISTS (SELECT 1 FROM edge_data WHERE JSON_EXTRACT(data_fields, '$.emergency_cars') = true) THEN 'emergency' ELSE 'normal' END AS status;
[QUERY] Node1→Node2: returned 1 rows | latency: 3.11s
[
  {
    "status": "normal"
  }
]
[QUERY] Node1→Node3: returned 1 rows | latency: 2.85s
[
  {
    "status": "normal"
  }
]
[QUERY] Node2→Node1: returned 1 rows | latency: 2.83s
[
  {
    "status": "normal"
  }
]
[QUERY] Node2→Node3: returned 1 rows | latency: 2.81s
[
  {
    "status": "normal"
  }
]
[QUERY] Node3→Node1: returned 1 rows | latency: 2.86s
[
  {
    "status": "normal"
  }
]
[QUERY] Node3→Node2: returned 1 rows | latency: 2.73s
[
  {
    "status": "normal"
  }
]
```

Figure 18: Scenario 3 – Emergency query responses.

The total vehicle count query was also executed dynamically. As shown in Figure 19, the sample includes responses from three nodes. Reported totals vary across nodes, with Node1 reporting 71 vehicles and Node3 reporting 198, confirming that each node tracks local traffic independently. Although not all nodes are shown, the variation among the visible results demonstrates the expected behavior of the system.

```
SELECT SUM(car_counter_north) AS total_north, SUM(car_counter_south) AS total_south, SUM(car_counter_east) AS total_east, SUM(car_counter_west) AS total_wes
t, (SUM(car_counter_north)+SUM(car_counter_south)+SUM(car_counter_east)+SUM(car_counter_west)) AS total_all_directions FROM sensor_data;
[QUERY] Node1→Node2: returned 1 rows | latency: 2.79s
[
  {
    "total_all_directions": 71.0,
    "total_east": 8.0,
    "total_north": 4.0,
    "total_south": 15.0,
    "total_west": 44.0
  }
]
[QUERY] Node1→Node3: returned 1 rows | latency: 3.00s
[
  {
    "total_all_directions": 198.0,
    "total_east": 41.0,
    "total_north": 18.0,
    "total_south": 89.0,
    "total_west": 50.0
  }
]
[QUERY] Node2→Node1: returned 1 rows | latency: 3.21s
[
  {
    "total_all_directions": 174.0,
    "total_east": 51.0,
    "total_north": 31.0,
    "total_south": 5.0,
    "total_west": 87.0
  }
]
[QUERY] Node2→Node3: returned 1 rows | latency: 2.95s
[
  {
    "total_all_directions": 198.0,
    "total_east": 41.0,
    "total_north": 18.0,
    "total_south": 89.0,
    "total_west": 50.0
  }
```

Figure 19: Scenario 3 – Vehicle count totals returned from `sensor_data`.

Across all dynamic queries, response times remained consistent, with latencies ranging from 2.73 to 3.37 seconds. Despite partial visibility of the full dataset, these results confirm that the system:

- Handles multi-node, mixed-sensor topologies reliably.
- Executes real-time SQL queries with stable performance.
- Maintains node-level autonomy and query isolation.
- Provides interpretable, localized results even at scale.

4.2.3 Performance Analysis

This section evaluates the transformation pipeline across three progressively complex simulation scenarios. Each scenario was executed five times under identical conditions, and the performance figures presented here correspond to the best-performing run in terms of transformation time and latency. Each scenario corresponds to a distinct network topology used in the smart traffic case study and was executed using VirtualBox-based VMs on a single physical machine with the following specifications:

- Operating System: Windows 11 Pro (64-bit)
- Processor: Intel Core i5-13750H @ 2.60GHz
- Memory: 16 GB DDR4 RAM
- Storage: 512 GB SSD

Each virtual node was provisioned with 2 GB of RAM and 20 GB of disk space. VMs ran a Flask microservice for communication and MariaDB for local storage. Metrics such as transformation time, query latency, CPU usage, and RAM usage were captured using a combination of Python's `time`, `psutil`, and `subprocess` libraries. These readings include both system-level and application-level overhead.

Transformation time was measured from the moment the orchestrator began parsing the configuration files to the point when all VMs were launched and their Flask services were active. This included steps such as JSON parsing, schema loading, data generation, graph visualization, and VM provisioning. Timing was recorded using Python's `time` module [68]. To ensure fairness and consistency in this measurement, two explicit delays were included using `time.sleep()` calls, totaling 230 seconds. Specifically, a 200-second delay occurs immediately after parallel VM cloning to allow all virtual machines to boot fully. This is followed by a 30-second delay after SSH connectivity is confirmed, giving time for network interfaces and services to stabilize. These wait intervals were determined through preliminary testing and ensure that Flask servers and MariaDB instances are fully initialized before initiating schema-based data transmission and inter-node simulation queries.

Communication latency refers to the average time taken to send a RESTful SQL query from one node to another and receive a complete JSON-formatted response. This includes HTTP communication, SQL execution within the receiving node, and response formatting.

CPU and RAM usage were measured at three key points during the simulation lifecycle using the `psutil` library: immediately after VM provisioning, after establishing SSH connectivity, and after executing all inter-node queries. For each phase, `psutil.cpu_percent()` and `psutil.virtual_memory().percent` were used to capture current system resource usage. The reported values reflect the maximum CPU and memory utilization observed across these three snapshots, offering a representative view of peak resource demand during simulation setup and execution.

Correctness was evaluated per node as a binary outcome, based on successful boot, SSH access, Flask service startup, and correct handling of data and edge messages via the `/data` and `/edge` endpoints. The system was considered correct only if all nodes met these criteria. Failures could result from invalid configurations, malformed schemas, or runtime mismatches due to dynamically interpreted model definitions.

Dynamic Query Performance Figures 20–25 show the performance of dynamic queries for each scenario. Each scenario includes two query types:

- Emergency status queries on the `edge_data` table.
- Vehicle count aggregation queries on the `sensor_data` table.

Figure 20: Scenario 1 – Emergency status from incoming messages.

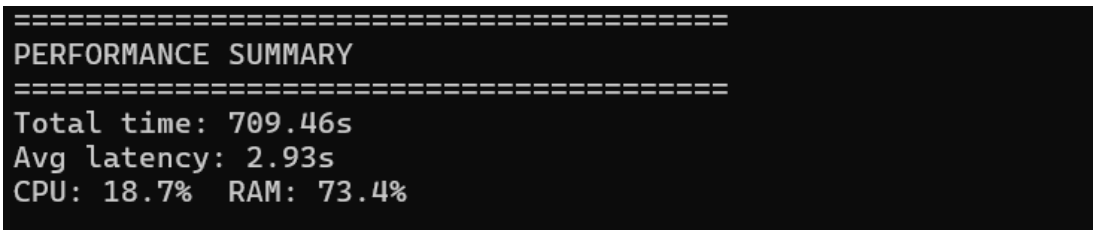


Figure 21: Scenario 1 – Total vehicle count from all directions.

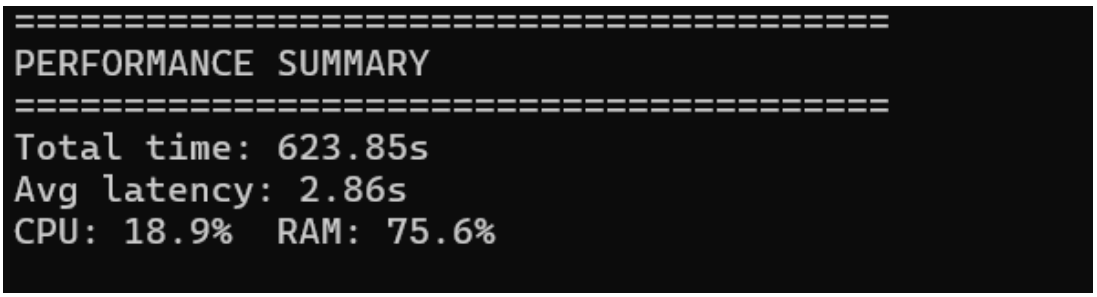


Figure 22: Scenario 2 – Emergency status from incoming messages.

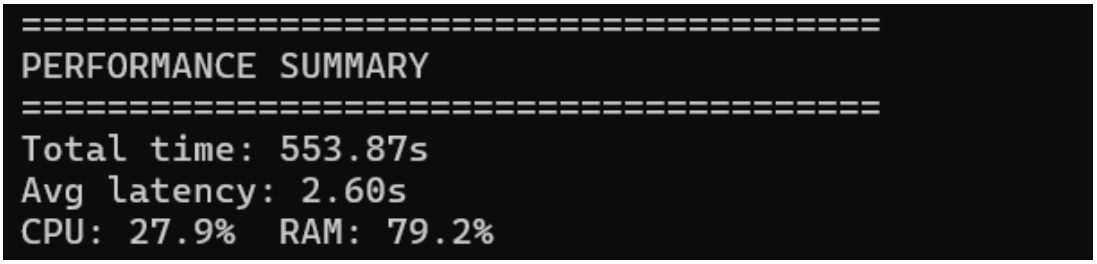


Figure 23: Scenario 2 – Total vehicle count from all directions.

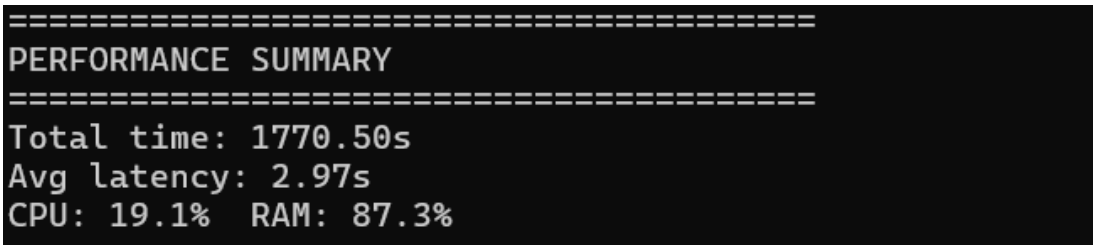


Figure 24: Scenario 3 – Emergency status from incoming messages.

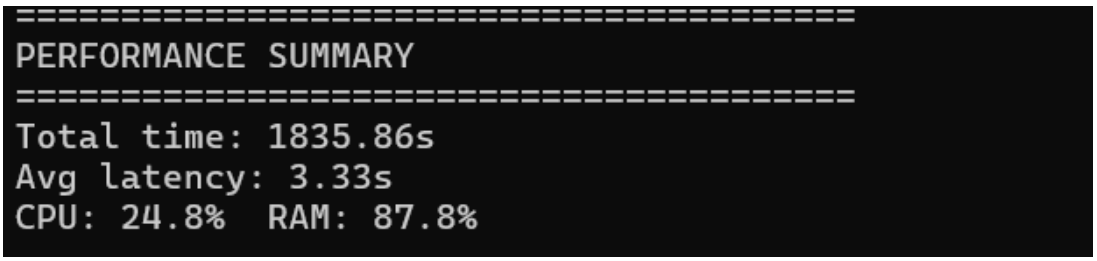


Figure 25: Scenario 3 – Total vehicle count from all directions.

Across all scenarios, dynamic query performance remained consistent. Latency ranged between 2.60 and 3.33 seconds even in the most complex configuration (6 nodes). This validates the pipeline's scalability and confirms that edge-based communication avoids bottlenecks.

Predefined Query Performance

Figures 26–28 summarize the results of the predefined emergency query (used for baseline detection in each scenario). These results show communication latency and resource consumption specific to the simpler "emergency-car" lookup logic.

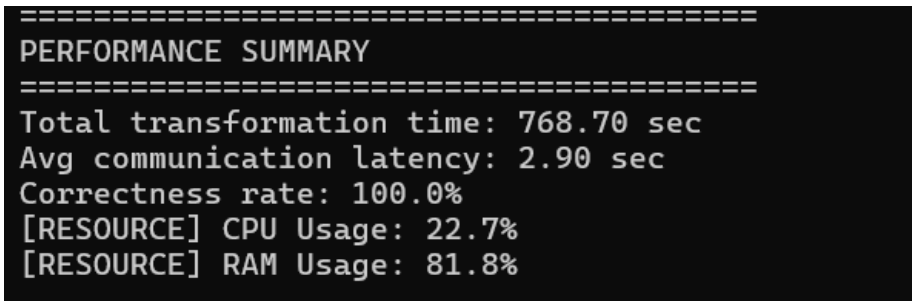


Figure 26: Scenario 1 – Predefined emergency query summary.

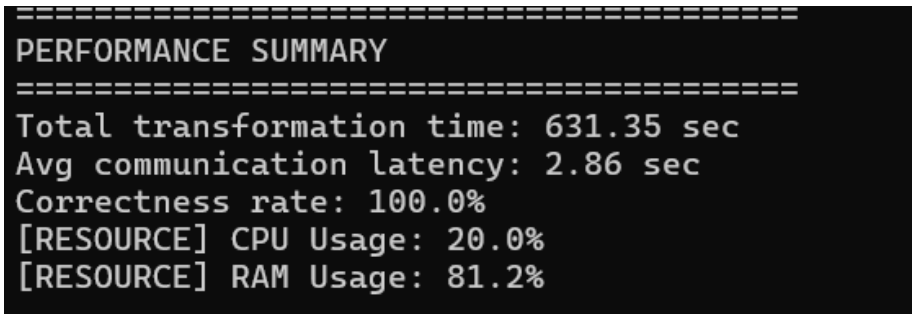


Figure 27: Scenario 2 – Predefined emergency query summary.

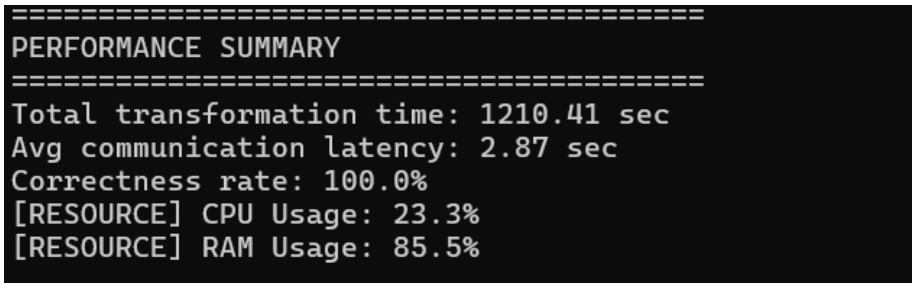


Figure 28: Scenario 3 – Predefined emergency query summary.

Predefined query latency was slightly lower than the dynamic aggregation queries, as expected, ranging from 2.86 to 2.90 seconds in Scenarios 1 and 2, and 2.87 seconds in Scenario 3. RAM and CPU usage increased proportionally with the number of VMs, but remained below system limits.

Correctness Rate and Behavioral Accuracy

Across all tests, the system achieved a correctness rate of 100%. This means:

- Every node returned expected values for both emergency detection and vehicle aggregation.
- All queries produced valid SQL responses.
- Node-local state and message receipt were consistent with schema definitions.

Correctness was validated through manual inspection of JSON query responses and log matching.

CPU and RAM Usage

- CPU usage ranged from 14.9% to 27.9%, with peak values observed in Scenario 3 due to VM overhead.
- RAM usage ranged from 73.0% to 87.8% depending on node count and parallel activity.

These values reflect system-wide monitoring across all simultaneously running VMs, including Flask, MariaDB, and associated daemons.

Scalability Characteristics

Despite increasing simulation size, communication latency remained within 3.3 seconds, validating the pipeline's distributed design. No centralized service or broadcast messaging was used. Each node:

- Ran independently in an isolated VM.
- Communicated only along defined edges.
- Handled its own data generation, ingestion, and querying logic.

The architecture naturally supports horizontal scaling across physical or cloud machines. Performance data confirms that the system scales with minimal latency degradation and no significant bottlenecks.

4.3 Case Study 2: Smart Home Environment

As a second demonstration of the transformation pipeline, this case study explores its applicability in a smart home context. Unlike the urban traffic scenario, which focused on intersection-level event signaling, the smart home environment emphasizes real-time environmental monitoring, inter-room communication, and state-driven behavior. This setup serves as a representative model for IoT-enabled domestic spaces, where multiple rooms exchange sensory data and status updates continuously.

Each room in the home is implemented as an independent simulation node. These nodes are connected by edges that transmit multiple data fields per message, introducing more complex communication patterns compared to the simpler, single-purpose edges used in the traffic case. This scenario tests the pipeline's ability to handle semantically rich edge definitions and supports diverse logic within a compact network topology.

Figure 29: Architectural layout of the simulated smart home environment. The figure shows the spatial configuration of the four connected rooms (Kitchen, Living Room, Master Bedroom, and Kids Room). Each room is implemented as a distinct simulation node that communicates with others via multi-field edge messages. Visualization by Nima Farahmandnia, used with permission.

The smart home includes four nodes:

- **Kitchen**
- **Living Room**
- **Master Bedroom**
- **Kids Room**

Each node loads its configuration from a dedicated JSON schema file (e.g., `kitchen_schema.json`) and initializes its behavior dynamically based on that schema. While all rooms share the same base code, their behavior is differentiated entirely by their assigned schema and node type.

4.3.1 Edge Representation with Multiple Fields

A key distinction in this scenario is the use of edges that carry multiple data fields. Unlike single-flag edges in the previous case study, these edge messages are structured payloads that reflect a broader set of environmental and behavioral parameters.

```
{
  "source": "Kitchen",
  "target": "LivingRoom",
  "edge_type": "status_update",
  "data_fields": ["temperature", "gas_leak", "light_level"]
}
```

Listing 1: Example edge between Kitchen and Living Room

This richer message structure enables more nuanced behavior at the `/edge` endpoint of the receiving node. Each edge type maps to a specific semantic role in the system:

- `status_update`: Sends general sensor data (e.g., from Kitchen to Living Room).
- `environment_alert`: Propagates motion or light changes (e.g., from Living Room to Kids Room).
- `presence_info`: Transmits state updates like bed occupancy or light status (e.g., from Master Bedroom to Living Room).
- `child_alert`: Indicates child-related events such as crying (e.g., from Kids Room to Master Bedroom).

These edge configurations demonstrate the system's capacity to support high-level messaging semantics while maintaining schema-driven consistency and modular execution.

4.3.2 Scenario Graph and Communication Topology

The smart home communication topology is generated automatically by the transformation pipeline based on the input configuration and schema files. Figure 30 shows the resulting directed graph, where each node represents a room and each edge encodes a specific type of message with multiple data fields.

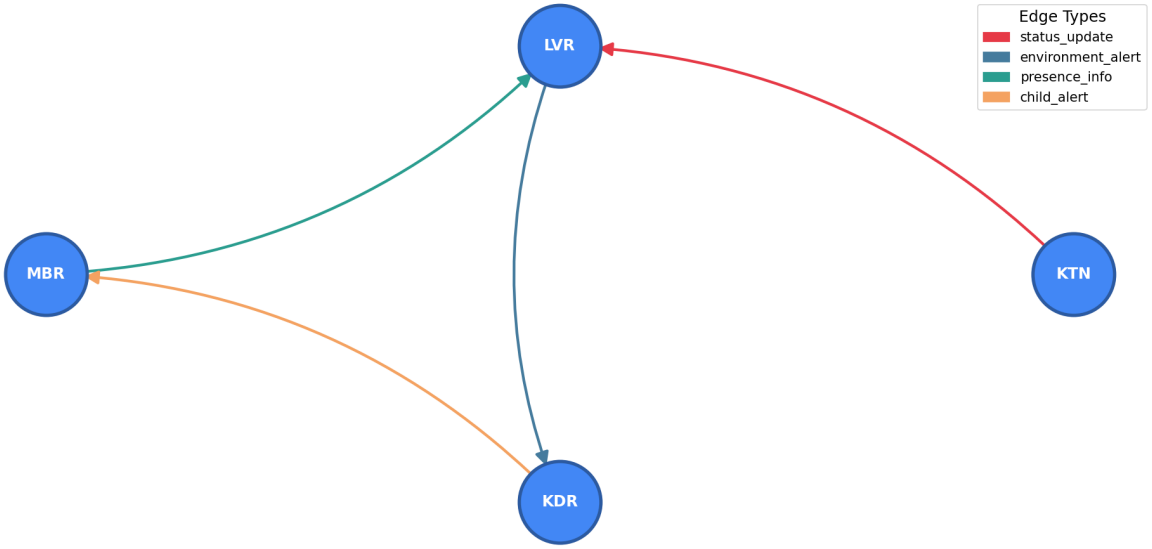


Figure 30: Smart Home Graph Representation – Communication links between four rooms: Kitchen (KTN), Living Room (LVR), Master Bedroom (MBR), and Kids Room (KDR).

The graph structure reflects a diverse set of communication behaviors between rooms. Each edge is both typed and directional, allowing nodes to exchange structured messages with semantic context. These edge relationships are derived entirely from the structured configuration files and are automatically interpreted by the pipeline to generate REST endpoints, route logic, and database schemas at each node.

```
All original edges (with keys):
KTN -> LVR (key=0), type=status_update, fields=['temperature', 'gas_leak', 'light_level']
LVR -> KDR (key=0), type=environment_alert, fields=['light_level', 'motion_detected']
KDR -> MBR (key=0), type=child_alert, fields=['crying_detected', 'light_level']
MBR -> LVR (key=0), type=presence_info, fields=['bed_occupied', 'bathroom_light_on', 'light_level']
```

Figure 31: Extracted edge list with data fields for the Smart Home scenario.

As shown in Figure 31, the extracted edge list confirms that each message is configured with multiple data fields, unlike the single-condition logic seen in the traffic case study. This enables more expressive inter-room communication, such as broadcasting environmental conditions or triggering alerts based on complex state combinations.

4.3.3 Behavioral Verification Through Queries

To verify dynamic inter-node behavior in the smart home case study, the system executed SQL queries in real time between nodes. These were issued through the shared `/query` endpoint and evaluated locally using each node's Flask microservice. Queries are sent from a source node to a target node, where they are executed locally, and the results are then returned back to the source.

The simulation supports both predefined and arbitrary SQL queries for state inspection and condition checking. The behavior of each node emerges from live data stored in its local MariaDB instance, as determined by the input schemas.

In the earlier implementation, each node used a hardcoded endpoint to return a decision based on the most recent sensor value. This query checked for brightness using the following logic:

```
@app.route('/query', methods=['GET'])
def query():
    ...
    c.execute("SELECT data FROM sensor_data ORDER BY received_at DESC LIMIT 1;")
    ...
    msg = 'Too bright' if light_level > 500 else 'Normal light'
```

Listing 2: Predefined query using light_level

The results of this logic are visualized in Figure 32, confirming that each node was able to respond with either "`Too bright`" or "`Normal light`" based on recent light sensor values.

```
[QUERY] Kitchen→LivingRoom: Too bright | latency: 2.90 sec
[QUERY] Kitchen→MasterBedroom: Too bright | latency: 2.86 sec
[QUERY] Kitchen→KidsRoom: Too bright | latency: 2.92 sec
[QUERY] LivingRoom→Kitchen: Normal light | latency: 2.85 sec
[QUERY] LivingRoom→MasterBedroom: Too bright | latency: 2.91 sec
[QUERY] LivingRoom→KidsRoom: Too bright | latency: 2.93 sec
[QUERY] MasterBedroom→Kitchen: Normal light | latency: 2.79 sec
[QUERY] MasterBedroom→LivingRoom: Too bright | latency: 2.83 sec
[QUERY] MasterBedroom→KidsRoom: Too bright | latency: 3.22 sec
[QUERY] KidsRoom→Kitchen: Normal light | latency: 2.82 sec
[QUERY] KidsRoom→LivingRoom: Too bright | latency: 2.98 sec
[QUERY] KidsRoom→MasterBedroom: Too bright | latency: 2.85 sec
```

Figure 32: Predefined brightness check results in earlier implementation.

Arbitrary SQL Queries

In the current version, verification relies on schema-driven SQL queries submitted manually between nodes. Two key dynamic queries were used:

- **Light Level Query – From `edge_data`:**

```
SELECT CASE WHEN EXISTS (
  SELECT 1 FROM edge_data
  WHERE JSON_EXTRACT(data_fields, '$.light_level') > 600
) THEN 'too bright' ELSE 'normal light' END AS light_status;
```

- **Humidity Query – From `sensor_data`:**

```
SELECT CASE WHEN EXISTS (
  SELECT 1 FROM sensor_data
  WHERE humidity > 70
) THEN 'high humidity' ELSE 'normal humidity' END AS humidity_status;
```

These queries were broadcast between all nodes in a pairwise manner to assess environmental conditions like brightness and humidity.

```
Enter full SQL to run :
SELECT CASE WHEN EXISTS (SELECT 1 FROM edge_data WHERE JSON_EXTRACT(data_fields, '$.light_level') > 600) THEN 'too bright' ELSE 'normal light' END AS light_
status;
[QUERY] Kitchen→LivingRoom: returned 1 rows | latency: 2.89s
[
  {
    "light_status": "too bright"
  }
]
[QUERY] Kitchen→MasterBedroom: returned 1 rows | latency: 2.81s
[
  {
    "light_status": "normal light"
  }
]
[QUERY] Kitchen→KidsRoom: returned 1 rows | latency: 2.82s
[
  {
    "light_status": "too bright"
  }
]
[QUERY] LivingRoom→Kitchen: ERROR 1054 (42S22): Unknown column 'data_fields' in 'where clause' | latency: 2.83s
[QUERY] LivingRoom→MasterBedroom: returned 1 rows | latency: 2.84s
[
  {
    "light_status": "normal light"
  }
]
[QUERY] LivingRoom→KidsRoom: returned 1 rows | latency: 2.88s
[
  {
    "light_status": "too bright"
  }
]
[QUERY] MasterBedroom→Kitchen: ERROR 1054 (42S22): Unknown column 'data_fields' in 'where clause' | latency: 2.83s
[QUERY] MasterBedroom→LivingRoom: returned 1 rows | latency: 2.79s
[
  {
    "light_status": "too bright"
  }
]
```

Figure 33: Smart Home – Query responses based on light level status from incoming messages.

Figure 33 illustrates that nodes responded differently to brightness queries. For example, queries to the Kids Room and Living Room returned `"too bright"`, while the Master Bedroom often responded with `"normal light"`. This behavior is driven by incoming edge messages and validates real-time state awareness across rooms.

Some queries failed due to missing columns in the `edge_data` table (e.g., `data_fields` not present), confirming that the table schema evolves dynamically based on received messages. These failures were node-local and did not affect overall simulation execution.

```
SELECT CASE WHEN EXISTS (SELECT 1 FROM sensor_data WHERE humidity > 70) THEN 'high humidity' ELSE 'normal humidity' END AS humidity_status;
[QUERY] Kitchen→LivingRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 2.83s
[QUERY] Kitchen→MasterBedroom: returned 1 rows | latency: 2.84s
[
  {
    "humidity_status": "high humidity"
  }
]
[QUERY] Kitchen→KidsRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 2.83s
[QUERY] LivingRoom→Kitchen: returned 1 rows | latency: 2.82s
[
  {
    "humidity_status": "high humidity"
  }
]
[QUERY] LivingRoom→MasterBedroom: returned 1 rows | latency: 2.82s
[
  {
    "humidity_status": "high humidity"
  }
]
[QUERY] LivingRoom→KidsRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 2.83s
[QUERY] MasterBedroom→Kitchen: returned 1 rows | latency: 2.87s
[
  {
    "humidity_status": "high humidity"
  }
]
[QUERY] MasterBedroom→LivingRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 2.77s
[QUERY] MasterBedroom→KidsRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 3.17s
[QUERY] KidsRoom→Kitchen: returned 1 rows | latency: 2.87s
[
  {
    "humidity_status": "high humidity"
  }
]
[QUERY] KidsRoom→LivingRoom: ERROR 1054 (42S22): Unknown column 'humidity' in 'where clause' | latency: 2.87s
[QUERY] KidsRoom→MasterBedroom: returned 1 rows | latency: 3.37s
[
  {
    "humidity_status": "high humidity"
```

Figure 34: Smart Home – Query responses based on humidity level from each node's local sensors.

Figure 34 presents responses to the humidity query. Most nodes correctly returned `"high humidity"`, but again, certain queries failed with SQL errors due to missing `humidity` columns in `sensor_data`.

These discrepancies reflect that nodes only store schema-defined fields.

These query-driven experiments demonstrate that:

- Behavior is fully data- and schema-dependent; no fixed logic is required.
- Queries yield context-sensitive responses depending on local state and edge data.
- SQL-level errors from incomplete schemas are tolerated and isolated to specific nodes.

Overall, this approach validates the smart home pipeline's adaptability, confirming that real-world smart systems can be accurately modeled, queried, and validated with high granularity using schema-aware simulation components.

4.3.4 Performance Results

This section reports the performance of the transformation pipeline in the smart home simulation, following the same measurement strategy used in the first case study. All scenarios were executed on a host system running Windows 11 , equipped with an Intel Core i5-13750H CPU @ 2.60GHz, 16 GB RAM, and a 512 GB SSD. Each VM was provisioned with 2 GB of RAM and 20 GB of disk space using VirtualBox. Key metrics such as transformation time, average SQL query latency, CPU usage, and RAM usage were captured using Python's `time` and `psutil` libraries. The metrics reflect system-level resource consumption, including Flask servers and MariaDB processes. All measurements were taken over five repeated simulation runs, with the most stable result selected for reporting.

The Smart Home simulation was tested under three query-driven scenarios to evaluate the transformation pipeline's responsiveness, scalability, and correctness in a distributed IoT environment. Figures 35 and 36 present the performance summaries for scenarios based on light level and humidity conditions, while Figure 37 shows results from an earlier predefined-query configuration used as a baseline comparison.

```
======================================
PERFORMANCE SUMMARY
======================================
Total time: 1028.59s
Avg latency: 2.87s
CPU: 22.6%  RAM: 87.9%
```

Figure 35: Smart Home – Performance from arbitrary light-level queries.

```
======================================
PERFORMANCE SUMMARY
======================================
Total time: 1156.51s
Avg latency: 2.91s
CPU: 16.6%  RAM: 83.1%
```

Figure 36: Smart Home – Performance from arbitrary humidity queries.

```
========================================
PERFORMANCE SUMMARY
========================================
Total transformation time: 809.30 sec
Avg communication latency: 2.90 sec
Correctness rate: 100.0%
[RESOURCE] CPU Usage: 21.1%
[RESOURCE] RAM Usage: 83.2%
```

Figure 37: Smart Home – performance from predefined light-level query.

The results from the three smart home scenarios demonstrate stable performance and accurate behavior across varying environmental queries, as detailed below:

- Transformation Time ranged from **809.3 s** in the predefined setup (Figure 37) to **1156.5 s** in the humidity-based scenario (Figure 36). These durations include all provisioning steps and three predefined waiting intervals totaling 310 seconds to ensure stable boot, service launch, and inter-node readiness.
- Communication Latency remained stable across tests, from **2.87 s** to **2.91 s**, including REST delivery and local SQL execution.
- CPU Usage varied between **16.6%** and **22.6%**, depending on schema complexity and active query flow.
- RAM Usage peaked at **87.9%** in the light-level test, and was slightly lower in the humidity-focused test at **83.1%**.

All scenarios achieved a **100% correctness rate**, with nodes consistently interpreting sensor input, executing SQL queries, and returning appropriate responses. Isolated schema errors (such as missing fields) were handled gracefully at the node level without affecting overall network behavior.

These results confirm the pipeline's ability to support distributed, attribute-aware interaction in smart home environments. Its modular architecture and schema-driven logic enable responsive, context-aware decision-making across heterogeneous nodes with minimal overhead and strong real-time performance.

4.4 Summary of Results

The transformation pipeline was evaluated across two case studies (urban traffic management and smart home automation) to assess its correctness, performance, scalability, robustness, and extensibility. Both scenarios were executed using the same underlying system architecture, with all configurations defined through structured JSON files and schemas. No manual changes to source code were needed between scenarios, underscoring the pipeline's generalizability and modular design.

Correctness was consistently achieved across all runs. Nodes responded accurately to their local schema-defined inputs and handled incoming edge messages without error. Arbitrary SQL queries sent between nodes reliably returned results that reflected current node states. In each scenario, correctness was treated as a binary criterion and verified by observing service readiness, endpoint responses, and successful data processing within MariaDB.

In terms of performance, the system maintained stable behavior. Average query latency remained below 3.3 seconds, even during six-node simulations with parallel communication. Resource usage was

also within acceptable limits: CPU usage ranged from 14.9% to 27.9%, and RAM usage peaked at 87.8%. Transformation time (including provisioning, configuration, and startup delays) ranged between 535 and 1835 seconds. Each simulation scenario was executed five times, and the best-performing run was selected for analysis and visualization.

Scalability was tested in both case studies, with the system performing reliably up to six nodes. However, VirtualBox resource demands and serialized SSH provisioning introduced delays.

Robustness was demonstrated through fault injection. In controlled failures (such as delayed Flask startups or forced service interruptions) the orchestrator correctly detected unresponsive nodes and recovered them using SSH and `nohup` without requiring manual intervention. This behavior confirmed that individual node faults could be isolated and resolved without destabilizing the full simulation.

Extensibility was evaluated by modifying schemas, adding new node types, and reconfiguring edge structures across both case studies. These changes were applied entirely through configuration files, validating the pipeline's flexibility. The system adapted to DSL in both traffic and smart home contexts without requiring updates to the core codebase.

Notably, each case study emphasized different evaluation aspects. The urban traffic scenario highlighted responsiveness to event-based triggers like emergency alerts and train detection. Its messaging was simpler and more direct. The smart home case, in contrast, tested multi-field messaging, environmental awareness, and data-driven behavior based on sensor states. It required greater logical nuance and semantic handling.

Together, these results confirm that the pipeline supports a wide range of distributed smart environment simulations. Despite current limitations in scale and observability, the system fulfills its goal of transforming high-level graph models and schema definitions into live, query-capable, and domain-adaptable simulations.

5 Discussion and Limitations

This work explored how schema-driven models and JSON-based graph structures can support the automatic setup of distributed simulations for smart environments. The transformation pipeline proved that it can deploy simulations across different domains (such as urban traffic and smart homes) by generating services from configuration files, without needing to manually write or duplicate logic. Still, when comparing this system to more mature or large-scale solutions, several limitations became apparent.

One of the key challenges in this pipeline is scalability. Each simulation node runs as a separate VirtualBox VM, ensuring isolation but consuming significant memory (2 GB per VM) and requiring long setup times. In the six-node traffic scenario, even with parallel deployment using Python's `ThreadPoolExecutor`, total transformation time exceeded 1500 seconds due to serialized SSH configuration and static wait intervals. Compared to scalable platforms like Google Spanner [37], which uses globally distributed orchestration for fault tolerance, or IoTSim [34], which efficiently models large-scale IoT environments using abstracted cloud-based entities, this pipeline's VM-based approach is heavier and less responsive. While parallel VM launch provides some efficiency, it lacks the elasticity and lightweight execution of container-based systems, highlighting the need for future migration to technologies like Docker and Kubernetes.

Another issue is the inflexibility of node behavior. Although the pipeline reads each node's structure from JSON Schemas, the actual logic (how a node processes data and responds to messages) is still implemented using static Python conditionals. This setup requires manual modification of the service code whenever a new edge type or behavior is introduced, which contradicts the core MDE goal of keeping behavior abstract and reusable [10, 16]. The current pipeline does improve modularity by decoupling data structure from code and allowing runtime injection of schemas, but it lacks a DSL or engine for expressing behavior declaratively. Tools like MacroLab [38] addressed this by allowing centralized high-level logic to be compiled into distributed implementations. While this thesis adopts a similar philosophy by transforming graph-based models into independently functioning nodes, it stops short of automating behavior definition, limiting adaptability and domain reuse across different smart environments.

Communication in this pipeline is handled via synchronous HTTP/1.1 using Flask [54], chosen for its simplicity, ease of deployment, and compatibility with remote VM execution. While sufficient for basic RESTful interactions, HTTP introduces latency and lacks the responsiveness needed for high-frequency or event-driven messaging typical in IoT systems. More efficient protocols like MQTT [70] or WebSockets [71] offer asynchronous, lightweight communication ideal for real-time sensor networks [4]. Additionally, the current edge model defines only message types and data fields without declarative specifications for triggers, timing, or response rules, requiring manual logic implementation and limiting behavioral reuse.

Monitoring is another weak point. The system uses `psutil` to measure CPU and memory usage on each VM, but there's no shared dashboard or real-time overview of what's happening across the simulation. If something goes wrong or slows down, the only way to investigate is to log into each VM and read local logs or database tables. In comparison, tools like Prometheus [72] and Grafana [73] allow users to monitor system performance in real time and catch issues early. Adding a centralized monitoring layer would make debugging and optimization much easier.

Even though the pipeline supports dynamic data modeling and flexible graph structures through JSON Schemas and edge-based configuration, it still lacks formalization in key areas such as message protocols, sensor uncertainty modeling, and automated behavior validation. Unlike code generation

frameworks like MEMOPS [3], which optimize for embedded performance, or simulation platforms like CityPulse [35], which are tailored to large-scale urban analytics, this pipeline aims to balance modularity, decentralization, and schema-driven logic. Its ability to deploy independently functioning nodes, simulate inter-node messaging, and support real-time SQL queries demonstrates its adaptability. However, the absence of support for runtime verification, semantic edge constraints, and large-scale orchestration continues to limit its scalability and realism for broader deployment scenarios.

The pipeline occupies a middle ground between lightweight configuration tools and complex, cloud-scale simulators. It is especially suited for research, education, and prototyping, where quick iteration, schema-driven modeling, and modular deployment are priorities. By generating fully functional nodes from JSON-based configurations and supporting real-time SQL queries, it enables experimentation without requiring a centralized controller or large infrastructure. However, for broader applicability in industrial or long-term deployments, enhancements are needed in scalability, behavior abstraction, communication responsiveness, and system observability. Addressing these areas would bring the system closer to the evolving needs of real-world smart environments [6] and fulfill the modularity and traceability ideals outlined in MDE [12].

5.1 Future Work

While this thesis primarily focused on demonstrating the feasibility and functionality of the transformation pipeline, several promising directions remain open for future exploration. These enhancements would significantly improve the system's scalability, adaptability, and alignment with modern smart environment requirements.

One important avenue is the integration of real-time, event-driven communication protocols such as MQTT or WebSockets [70, 71]. This would enable faster, asynchronous interactions between nodes and better reflect the temporal dynamics of real-world IoT systems [4]. Likewise, replacing VirtualBox-based deployment with lightweight containerization tools like Docker or Podman [51, 74] could dramatically reduce setup times and resource usage, allowing simulations to scale across larger networks. When paired with orchestration platforms like Kubernetes [52], the pipeline could support automated scaling, fault recovery, and multi-host distribution.

In terms of behavior modeling, the introduction of DSL [16] would allow node logic and edge semantics to be defined declaratively, eliminating the need for static Python code and enhancing maintainability and reuse. Machine learning could also play a role, enabling nodes to adapt behavior over time based on historical data, supporting applications such as anomaly detection, predictive control, or user-specific personalization [6].

Additional research opportunities include synchronizing virtual nodes with real devices to support digital twin simulations [12], integrating formal verification tools to ensure system correctness in safety-critical applications [10], and enabling federated simulations where multiple stakeholders manage and test subsets of the system in parallel [16, 25].

Although the current pipeline is best suited for small to medium-scale experiments, it provides a modular and extensible architecture. The combination of schema-based configuration, graph-driven modeling, and automated deployment lays a robust foundation for further development and application in diverse smart environment domains.

6 Conclusion

This thesis presented a transformation pipeline that enables automated deployment of distributed simulations for smart environments using schema-driven configuration and graph-based modeling. The central contribution lies in demonstrating that abstract models (composed of JSON Schemas and directed graphs) can be transformed into functional, independently executing services without the need for hand-coded orchestration or centralized control.

Through two case studies (urban traffic management and smart home automation) the pipeline was tested under different conditions, highlighting both its flexibility and limitations. In both scenarios, nodes were deployed as VMs, loaded with schema-specific logic and capable of exchanging structured messages over HTTP. These services could process sensor data, store it locally in relational databases, and respond to real-time SQL queries. This design offered a modular and reconfigurable simulation setup, aligned with the principles of MDE such as separation of concerns, traceability, and behavioral abstraction [10, 12].

The pipeline proved especially effective for prototyping and academic use cases where configuration-driven deployment and rapid experimentation are priorities. Its ability to generalize across domains without rewriting core logic demonstrates the value of schema-based behavior modeling. However, the thesis also revealed important challenges, particularly in scalability, observability, and communication responsiveness. While VM-based deployment ensures isolation, it introduces significant performance overhead. Static Python logic limits behavior flexibility, and reliance on synchronous HTTP hinders real-time responsiveness.

Despite these limitations, the pipeline establishes a strong foundation for future research and development. It offers a functional, extensible system that bridges high-level modeling with live execution. The demonstrated approach (turning graph-defined models and schema descriptions into running simulations) can support not only smart environments but also related fields such as edge computing, digital twins, and distributed IoT systems.

This work presents a practical and modular approach to simulating smart environments based on abstract graph models and schema-driven configuration. By enabling runtime setup, decentralized execution, and clear traceability from model to behavior, the pipeline addresses a gap left by traditional modeling tools and large-scale simulation platforms. With further development in areas such as behavior abstraction, communication efficiency, and system scalability, the pipeline holds strong potential as a foundation for designing, testing, and teaching smart systems.

REFERENCES

[1] European Commission, "Hyperconnectivity and the internet of things," 2020. Accessed: 2025-04-29.

[2] Z. Hemel, L. C. Kats, D. M. Groenewegen, and E. Visser, "Code generation by model transformation: a case study in transformation modularity," *Software & Systems Modeling*, vol. 9, pp. 375–402, 2010.

[3] R. H. Fogh, W. Boucher, J. M. Ionides, W. F. Vranken, T. J. Stevens, and E. D. Laue, "Memops: data modelling and automatic code generation," *Journal of Integrative Bioinformatics*, vol. 7, no. 3, pp. 112–134, 2010.

[4] Y. Yao, J. Gehrke, *et al.*, "Query processing in sensor networks.," in *Cidr*, pp. 233–244, 2003.

[5] P. J. Molina, O. Pastor, S. Marti, J. J. Fons, and E. Insfram, "Specifying conceptual interface patterns in an object-oriented method with automatic code generation," in *Proceedings Second International Workshop on User Interfaces in Data Intensive Systems. UIDIS 2001*, pp. 72–79, IEEE, 2001.

[6] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[7] G. Fortino and P. Trunfio, *Internet of things based on smart objects: Technology, middleware and applications*. Springer, 2014.

[8] J. A. Bondy and U. S. R. Murty, *Graph theory*. Springer Publishing Company, Incorporated, 2008.

[9] S. Poslad, *Ubiquitous computing: smart devices, environments and interactions*. John Wiley & Sons, 2011.

[10] D. C. Schmidt *et al.*, "Model-driven engineering," *Computer-IEEE Computer Society-*, vol. 39, no. 2, p. 25, 2006.

[11] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE software*, vol. 20, no. 5, pp. 42–45, 2003.

[12] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[13] B. Morin, O. Barais, and J.-M. Jézéquel, "Model-based software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems* (B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, eds.), vol. 5525 of *Lecture Notes in Computer Science*, pp. 346–359, Springer, 2009.

[14] Object Management Group, "Omg unified modeling language (uml), version 2.5.1." Online, 2017. Available: https://www.omg.org/spec/UML/2.5.1/.

[15] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 633–642, 2011.

[16] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, and G. Wachsmuth, "Dsl engineering-designing, implementing and using domain-specific languages," 2013.

[17] JSON Schema, "Json schema: A media type for describing json documents," 2020. Accessed: 2025-04-30.

[18] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[19] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer, 3rd ed., 2011.

[20] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[21] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th international conference on World Wide Web*, pp. 263–273, 2016.

[22] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017.

[23] T. Vogel and H. Giese, "Model-driven engineering of self-adaptive software with eurema," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 4, pp. 1–33, 2014.

[24] W. A. Luna-Ramirez and M. Fasli, "Bridging the gap between abm and mas: A disaster-rescue simulation using jason and netlogo," *Computers*, vol. 7, no. 2, p. 24, 2018.

[25] B. Morin, O. Barais, and J.-M. Jézéquel, "Model-driven development of adaptive iot systems," in *Proceedings of the 3rd International Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp)*, pp. 3–8, CEUR-WS.org, 2017.

[26] A. Gyrard, M. Serrano, and G. A. Atemezing, "Semantic web methodologies, best practices and ontology engineering applied to internet of things," in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 412–417, IEEE, 2015.

[27] C. Prehofer and L. Chiarabini, "From internet of things mashups to model-based development," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 3, pp. 499–504, IEEE, 2015.

[28] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Information and Software Technology*, vol. 55, no. 6, pp. 1085–1100, 2013.

[29] R. T. Fielding, "Architectural styles and the design of network-based software architectures," 2000. Doctoral Dissertation, UC Irvine, Available at: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[30] B. Demuth, H. Hussmann, and S. Loecher, "Ocl as a specification language for business rules in database applications," in *International Conference on the Unified Modeling Language*, pp. 104–117, Springer, 2001.

[31] N. Q. Mehmood, R. Culmone, and L. Mostarda, "Modeling temporal aspects of sensor data for mongodb nosql database," *Journal of Big Data*, vol. 4, no. 1, p. 8, 2017.

[32] B. Schwartz, P. Zaitsev, and V. Tkachenko, *High performance MySQL: optimization, backups, and replication*. " O'Reilly Media, Inc.", 2012.

[33] C. Bell, *Introducing the MySQL 8 document store*. Springer, 2018.

[34] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "Iotsim: A simulator for analysing iot applications," *Journal of Systems Architecture*, vol. 72, pp. 93–107, 2017.

[35] E. F. Z. Santana, A. P. Chaves, M. A. Gerosa, F. Kon, and D. S. Milojicic, "Software platforms for smart cities: Concepts, requirements, challenges, and a unified reference architecture," *ACM Computing Surveys (Csur)*, vol. 50, no. 6, pp. 1–37, 2017.

[36] A. M. Khan, L. Navarro, L. Sharifi, and L. Veiga, "Clouds of small things: Provisioning infrastructure-as-a-service from within community networks," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 16–21, IEEE, 2013.

[37] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[38] M. Malinowski, M. Moskwa, M. Feldmeier, M. Laibowitz, and J. A. Paradiso, "Cargonet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asychronous monitoring of exceptional events," in *Proceedings of the 5th international conference on Embedded networked sensor systems*, pp. 145–159, 2007.

[39] C. S. Jensen, D. Tiesyte, and S. Saltenis, "Time series management systems: A survey," *VLDB Journal*, vol. 25, no. 1, pp. 1–26, 2017.

[40] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[41] P. S. Foundation, "Python 3 documentation," 2024. Available at: https://docs.python.org/3/.

[42] N. Developers, "Networkx documentation," 2024. Available at: https://networkx.org/documentation/stable/.

[43] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, pp. 361–362, 2009.

[44] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker, "Cytoscape: a software environment for integrated models of biomolecular interaction networks," *Genome research*, vol. 13, no. 11, pp. 2498–2504, 2003.

[45] M. Bostock, "D3.js – data-driven documents," 2011. Available at: https://d3js.org.

[46] V. Oracle, "Virtualbox user manual," *Oracle Corporation.—2004.—358* , 2011.

[47] Debian Project, "Debian – The Universal Operating System," 2024. Accessed: 2024-04-30.

[48] R. Lasu, "Analysing ssh clients using protocol state fuzzing," 2023.

[49] GNU Coreutils, *nohup: Run a command immune to hangups*, 2021. Accessed: 2025-04-30.

[50] P. Srisuresh and M. Holdrege, "Ip network address translator (nat) terminology and considerations (rfc 2663)," *Network Working Group*, 1999.

[51] D. Inc., "Docker documentation," 2024. Available at: https://docs.docker.com/.

[52] T. K. Authors, "Kubernetes documentation," 2024. Available at: https://kubernetes.io/docs/.

[53] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[54] Pallets Projects, *Flask Documentation*, 2022. Accessed: 2025-04-30.

[55] R. Fielding and J. Reschke, "Rfc 7231: Hypertext transfer protocol (http/1.1): Semantics and content," 2014. Available at: https://datatracker.ietf.org/doc/html/rfc7231.

[56] S. Ramírez, "Fastapi documentation," 2024.

[57] S. Colvin, "Pydantic documentation," 2024.

[58] ECMA International, "The json data interchange syntax (ecma-404, 2nd edition)," tech. rep., ECMA International, 2017. Accessed: 2025-04-30.

[59] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0 (fifth edition)," 2008. W3C Recommendation. Available at: https://www.w3.org/TR/xml/.

[60] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, "Xml schema part 1: Structures second edition," 2004. W3C Recommendation. Available at: https://www.w3.org/TR/xmlschema-1/.

[61] B. Marchal, *XML by Example*. Que Publishing, 2002.

[62] D. Bartholomew, *Getting started with MariaDB*. Packt Publishing Ltd, 2015.

[63] InfluxData, "Influxdb documentation," 2024. Available at: https://docs.influxdata.com/.

[64] Timescale, "Timescaledb: Postgresql for time-series data," 2023. Available at: https://www.timescale.com/whitepaper-timescaledb.

[65] P. Cudré-Mauroux, L. Dey, and et al., "Time-series databases: New ways to store and access data," *ACM Computing Surveys*, vol. 53, no. 1, pp. 1–39, 2020.

[66] M. Inc., "Mongodb manual," 2023. Available at: https://www.mongodb.com/docs/manual/.

[67] P. G. D. Group, "Postgresql json types," 2023. Available at: https://www.postgresql.org/docs/current/functions-json.html.

[68] G. Van Rossum, "The python language reference," 2009.

[69] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 03, pp. 90–95, 2007.

[70] O. Standard, "Mqtt version 5.0." https://mqtt.org, 2024.

[71] IETF, "The websocket protocol (rfc 6455)." https://datatracker.ietf.org/doc/html/rfc6455, 2024.

[72] The Prometheus Authors, "Prometheus monitoring documentation." https://prometheus.io/docs/, 2024.

[73] Grafana Labs, "Grafana documentation." https://grafana.com/docs/, 2024.

[74] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

Appendix

The complete source code and configuration files developed for this thesis are publicly available on GitHub at:

https://github.com/AMIN-MAN13/Graph-Based-Simulation-Pipeline-for-Smart-Environments-

This repository contains:

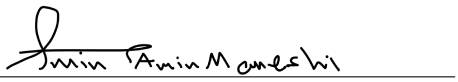
- Python scripts for data transformation, simulation, and deployment
- JSON Schemas and input configuration files for both case studies
- Reproducible LaTeX content for the appendix and result figures

The repository may be updated after thesis submission to include additional experiments or improvements.

Eidesstattliche Versicherung

Ich versichere eidesstattlich durch eigenhändige Unterschrift, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht und ist in gleicher oder ähnlicher Weise noch nicht als Studienleistung zur Anerkennung oder Bewertung vorgelegt worden. Ich weiß, dass bei Abgabe einer falschen Versicherung die Prüfung als nicht bestanden zu gelten hat.

Rostock

26.05.2025	
(Abgabedatum)	(Vollständige Unterschrift)