future internet

MDPI

Article

A Synergistic Elixir-EDA-MQTT Framework for Advanced Smart Transportation Systems

Yushan Li [1,2] and Satoshi Fujita [1,2,*]

[1] Graduate School of Advanced Science and Engineering, Hiroshima University, Higashi-Hiroshima 739-0046, Japan; yushanli433@gmail.com
[2] Department of Information Engineering, Hiroshima University, Higashi-Hiroshima 739-0046, Japan
* Correspondence: satoshi.fujita.g@gmail.com

Abstract: This paper proposes a novel event-driven architecture for enhancing edge-based vehicular systems within smart transportation. Leveraging the inherent real-time, scalable, and fault-tolerant nature of the Elixir language, we present an innovative architecture tailored for edge computing. This architecture employs MQTT for efficient event transport and utilizes Elixir's lightweight concurrency model for distributed processing. Robustness and scalability are further ensured through the EMQX broker. We demonstrate the effectiveness of our approach through two smart transportation case studies: a traffic light system for dynamically adjusting signal timing, and a cab dispatch prototype designed for high concurrency and real-time data processing. Evaluations on an Apple M1 chip reveal consistently low latency responses below 5 ms and efficient multicore utilization under load. These findings showcase the system's robust throughput and multicore programming capabilities, confirming its suitability for real-time, distributed edge computing applications in smart transportation. Therefore, our work suggests that integrating Elixir with an event-driven model represents a promising approach for developing scalable, responsive applications in edge computing. This opens avenues for further exploration and adoption of Elixir in addressing the evolving demands of edge-based smart transportation systems.

Keywords: Elixir; edge computing; event-driven architecture; concurrency; smart transportation

Citation: Li, Y.; Fujita, S. A Synergistic Elixir-EDA-MQTT Framework for Advanced Smart Transportation Systems. *Future Internet* **2024**, *16*, 81. https://doi.org/10.3390/fi16030081

Academic Editors: Yuezhi Zhou and Xu Chen

Received: 20 January 2024
Revised: 22 February 2024
Accepted: 25 February 2024
Published: 28 February 2024

1. Introduction

1.1. Background

With the rapid development of Internet of Things (IoT), an exponentially increasing number of smart devices are being connected, generating massive amounts of real-time data that need to be processed instantly. Traditional cloud computing architectures, relying on centralized data centers, are insufficient to meet the low-latency and location awareness requirements of many emerging IoT applications [1]. This has led to growing interest in edge computing, which pushes computation and data storage closer to the location where the data are generated. By processing data at the edge, latency can be reduced significantly while also decreasing bandwidth usage [2].

The global market for the Internet of Things was estimated to be worth around USD 182 billion in 2020 [3], and it is anticipated to triple in size by 2030, reaching over USD 621 billion. At the same time, according to a report by Grand View Research, the global edge computing market size is expected to reach USD 43.4 billion by 2027 [4], driven by the increasing adoption of IoT devices and the need for real-time data analysis and processing at the network edge.

However, existing edge computing solutions predominantly employ imperative programming languages like C/C++, Java, and Python, which incur complexity in developing and maintaining large applications. The tight coupling between components, lack of fault-tolerance mechanisms, and single-threaded execution models in these languages make

them ill suited for the dynamic and distributed nature of edge computing systems. To overcome these limitations, we propose the use of Elixir, a modern functional programming language built on the robust Erlang Virtual Machine, for building highly available and fault-tolerant applications for edge computing.

In our previous work [5], we conducted experiments comparing an Elixir-based message broker with an equivalent Rust implementation under different network conditions. The results validated Elixir's resilience and low latency, confirming its potential for edge computing deployments. Building on these findings, in this paper, we propose a novel programming framework in the edge computing paradigm. The framework combines the benefits of Elixir language, and event-driven architecture, with MQTT protocol. We demonstrate two use cases in the field of smart transportation applications: a traffic light system that optimizes traffic flow in the intersection, and a cab dispatch system that coordinates taxis and passengers based on real-time location data. The evaluation of the cab dispatch system shows the low latency and good performance of the system.

In this paper, we will start our exploration with the introduction. Subsequently, we will provide the essential properties of the language we use in Section 2. In Section 3, we will introduce our architecture design in detail. Two specific case studies are demonstrated in the next section. We next discuss the details of the prototype system in Section 5. Then, Section 6 is provided as the evaluation part. To conclude, we synthesize the key findings and implications of our research in the last section.

This paper is an extended version of a paper [5] presented at CANDAR 2023. The difference to the conference version is summarized as follows: (1) we add recent papers concerned with smart transportation systems and the application of event-driven architecture in smart cities as related work; (2) we add event-driven architecture as an important component in our proposed framework, and utilize it in a smart transportation application; (3) in the evaluation, we add various kinds of experiments to demonstrate the overall performance of our system in smart transportation scenarios; and (4) add a traffic light system for supplement the explanation for our proposed framework.

1.2. Related Research

This subsection overviews related research on this study, focusing on three research areas: event-driven architecture (EDA), smart transportation, and Elixir/Erlang-based systems.

1.2.1. Related Work Concerned with EDA

The adoption of event-driven architecture (EDA) in the development of smart cities has become a widely applied concept in recent research. This trend highlights the versatile application of EDA across various fields, demonstrating its potential to address a broad spectrum of challenges within the smart city paradigm. The diversity observed in these studies reflects EDA's flexibility and its capability to enhance systems in multiple domains, from healthcare, which directly impacts human health and safety, to urban traffic management, aiming at optimizing flow and increasing transportation efficiency. Although these approaches differ in their technical implementations, they share a core objective: leveraging advanced technology to improve human life, whether through safeguarding human health or promoting environmental sustainability.

This work by Amir Rahmani, Babaei, and Souri [6] introduced an event-driven IoT architecture for data analysis of reliable healthcare applications, including context, event, and service layers. Furthermore, the study presents complex event processing (CEP) as an innovative solution, integrating automated intelligence within the event layer to enhance the system's responsiveness and decision-making capabilities. This contrasts with our approach, which integrates Elixir and MQTT for an edge-based programming framework. While CEP offers advanced data processing capabilities ideal for healthcare applications, our methodology leverages Elixir's robust concurrency and MQTT's efficient message handling, tailored to the dynamic nature of smart transportation systems. This divergence highlights the adaptability of event-driven architectures across varying domains. The next

work by Behnam Khazael et al. [7] also utilized complex event processing (CEP) systems. Differently, it introduced Geo-TESLA, an advanced complex event processing language tailored for smart city applications, enhancing the detection and reporting of complex events within urban settings by leveraging spatial data types and operations.

Another work that combines event-driven architecture and smart city is the work by Garcia Alvarez, Manuel, Javier Morales, and Menno-Jan Kraak [8]. They offered an approach for spatiotemporal capabilities in information services for smart cities and developed a reference architecture of event-driven applications. This work demonstrates the feasibility, performance, and scalability of event-driven applications in real-time processing and detecting geographic events, leveraging IoT technologies. Xiao Changjiang [9] offered an event-driven focusing service (EDFS) method that uses cyberphysical infrastructures for emergency response in smart cities.

1.2.2. Related Work Concerned with Smart Transportation

With the development of smart cities, the integration of edge computing technologies plays an important role in transforming urban infrastructure. The adoption of edge computing not only facilitates real-time data processing at the network's edge, enhancing the efficiency and responsiveness of smart transportation applications, but also opens up new avenues for addressing complex challenges inherent in urban environments. This complexity is underscored by both the comprehensive survey by Saeik, Firdose et al. [10] on task offloading in edge and cloud computing and the detailed examination of resource scheduling strategies in edge computing by Luo et al. [11]. These works illustrate the diverse approaches and theoretical foundations developed to optimize task offloading processes and resource scheduling across different edge computing scenarios. The approach also aligns with the broader objectives of improving traffic flow, enhancing vehicular communication, and ensuring safety, thus contributing to the overall efficiency and sustainability of urban living.

The next part is within the smart transportation of edge computing realm. The first related work discussed a decision support method of event-driven architecture for a traffic management system [12]. This paper illustrates how event-driven architecture (EDA) and complex event processing are used for real-time processing and analysis of extensive data streams generated by sensors and vehicles. The core objective is real-time monitoring and control of traffic flow, exemplified in a smart traffic management system prototype in Bilbao, Spain. While the paper effectively demonstrates the use of event-driven architecture for traffic management, our research focuses on leveraging edge computing technologies. This approach significantly reduces communication latency and real-time responses, which is a crucial aspect in smart transportation systems. Wei-Hsun Lee et al. proposed a novel design and implementation of a smart traffic signal control (STSC) system that enhances vehicular communication and traffic management [13]. We were inspired by the design of the smart traffic signal control system; however, we used a different solution that combines Elixir and event-driven architecture to handle the vehicular communication in real time. The work by Ke Ruimin [14] focused on edge computing for real-time near-crash detection in smart transportation. It used IoT devices like Nvidia Jetson TX2 for processing video streams to identify near-crash events.

The research highlighted above offers diverse solutions and implementations for smart transportation systems. Differently, our study introduces a more novel approach by leveraging an edge-based framework using Elixir, combined with event-driven architecture and MQTT, to efficiently handle the real-time processing of huge volumes of data in intelligent transportation systems.

1.2.3. Related Work Concerned with Elixir/Erlang-Based Systems

In addition to research on event-driven architecture and smart transportation, there is also compelling evidence regarding the study of Erlang language compatibility and hardware adaptability in IoT systems. GRiSP is a hardware platform and a bare-metal

Erlang virtual machine designed for real-time embedded systems. Several research studies have attempted to build IoT systems using GRiSP. The papers [15,16] proposed a framework called Achlys, to realize general-purpose edge computing using only nodes on a sensor network without relying on gateways or connections to cloud servers. It offers great suitability for distributed applications in IoT edge networks Hera [17] is a Kalman-filter-based sensor fusion framework whose application programs are written in Erlang running on GRiSP. With this framework, high-level processing for asynchronous and fault-tolerant sensor fusion can be realized directly at the edge of the IoT network. Since GRiSP is bare metal Erlang, it has full compatibility with Elixir which runs on BEAM. We contemplate deploying GRiSP in actual environments in the subsequent phase of our research.

2. Elixir Programming Language

This section identifies three critical properties for implementing robust edge computing frameworks: *fault tolerance*, *real-time processing*, and *support for nondisruptive operation*. Fault tolerance ensures continuous operation despite individual component failures. Real-time processing guarantees timely data processing within specified latency constraints. The nondisruptive operation allows updates and maintenance without service interruptions. This section analyzes how Elixir, leveraging its foundation in Erlang, successfully embodies these crucial properties.

2.1. Erlang: A Foundation for Resilience

Erlang, introduced in 1986 by Ericsson, is a functional programming language designed for concurrent systems with the "run forever" philosophy [18]. This focus on robust, nonstop systems makes Erlang a natural choice for edge computing.

Several key features contribute to Erlang's suitability:

- Concurrent processes: Erlang runs multiple lightweight processes on the Erlang Virtual Machine (BEAM). Individual process failures are handled by automatic termination and restart, ensuring system resilience.
- Efficient resource allocation: Erlang's process scheduling ensures timely responsiveness for real-time tasks. Processes can migrate between execution queues, minimizing wait times and optimizing message exchange.
- Hot code loading: Updates can be applied without service interruptions via hot code loading, enabling nondisruptive operation and continuous maintenance.

These features demonstrate Erlang's strength in building reliable and responsive systems, making it a valuable foundation for edge computing frameworks.

2.2. Elixir: Building on Erlang's Legacy

Elixir, built on top of the BEAM virtual machine, inherits Erlang's core strengths. BEAM compiles Elixir code to bytecode for efficient execution. Lightweight processes and message-passing communication foster concurrency and fault tolerance, as failures in one process do not affect others. This inherent resilience is crucial for edge environments where reliability is paramount. Unlike imperative languages, like Java and C++, that rely on shared memory and heavyweight threads, Elixir's message-passing model avoids complex synchronization issues and performance bottlenecks associated with shared resource contention.

Beyond inheriting Erlang's strengths, Elixir offers additional advantages for building scalable and maintainable edge applications:

- Functional programming paradigm: Elixir encourages a side-effect-free programming style, where functions produce outputs solely based on their inputs, simplifying code comprehension and testing.
- Powerful tools and libraries: Elixir provides a rich ecosystem of libraries and tools designed for building robust and maintainable applications.

These combined benefits make Elixir a compelling choice for developing reliable and performant edge computing frameworks. The next subsection delves deeper into Elixir's programming model and its specific advantages for edge development.

2.3. Elixir's Programming Model

This subsection examines key aspects of Elixir's programming model that contribute to its suitability for developing edge computing frameworks: polymorphism, meta-programming, and code conciseness.

2.3.1. Polymorphism via Protocols

Both Elixir and Erlang achieve polymorphism through pattern matching and function dispatch. However, Elixir introduces the powerful concept of *protocols*, enhancing flexibility and intuitiveness. Protocols define a set of functions that any data type can implement, enabling generic operations for different types and implementations. Elixir dynamically recognizes and calls the corresponding specific implementation, demonstrating inherent polymorphism. Additionally, protocols can have "fallback to Any" mechanisms, providing default implementations for unknown types. This promotes code reusability and simplifies handling heterogeneous data structures.

2.3.2. Powerful Meta-Programming Capabilities

Meta-programming, the ability to manipulate and generate code at runtime [19], empowers Elixir development. Compared to Erlang, Elixir offers a more comprehensive and user-friendly meta-programming toolkit through its macro system. This system provides higher-level abstractions and richer functionalities, including module metadata, annotations, reflection, and code evaluation. Accessing the abstract syntax tree (AST) through macros facilitates powerful code transformations and generation, leading to increased development efficiency and improved code quality.

2.3.3. Concise and Expressive Functional Code

Elixir's functional features contribute significantly to code conciseness. Functional constructs like immutability and explicit function definitions enhance program clarity and control flow visualization. This is particularly beneficial for edge computing applications, where compact and understandable code is crucial for efficient execution and debugging. Furthermore, conciseness reduces development time and complexity, making Elixir a compelling choice for rapid development cycles.

2.4. Summary: Why Elixir for Edge Computing?

This section has identified three key features of Elixir's programming model that make it ideally suited for edge computing applications: robust polymorphism and protocol mechanisms, powerful meta-programming capabilities, and inherent code conciseness through functional idioms. These features, coupled with Elixir's rapidly growing library ecosystem, solidify its position as a top choice for building reliable and efficient edge computing frameworks.

3. Architecture Design

This section outlines the key principles and components of the proposed architecture for smart transportation edge computing. Details of the prototype implementation based on this architecture are presented in the succeeding sections. Our envisioned system continuously collects and stores sensor data from urban areas for efficient processing and response to user requests. It demands scalability, real-time functionality, and fault tolerance, aligning perfectly with the capabilities of the Elixir language, as discussed in the previous section.

The proposed architecture comprises multiple interacting components, designed for specific functionalities. Asynchronous message passing with MQTT for event transport and

Elixir for event processing facilitates concurrent execution and fault isolation. Specifically, Elixir's lightweight concurrency and distributed processing handle asynchronous events in a scalable manner (Section 3.2), while MQTT's publish–subscribe messaging distributes events across service components (Section 3.3).

3.1. Event-Driven Architecture for Edge Computing

Event-driven architecture (EDA) [20] centers around event production, detection, consumption, and reaction, where "event" signifies a significant state change. EDA excels in systems requiring real-time operations, asynchronous communication, and high scalability [20,21]. While traditional EDA often centralizes event handling [22,23], EDA for edge computing, like smart transportation, requires optimization for low-latency and local data processing to minimize network overhead and response time. In other words, edge computing tailors EDA to address its inherent challenges: real-time data processing, resource-constrained environments, and distributed computational nodes.

Prominent EDA systems include Kafka Streams [24], Azure Event Grid [25], and RabbitMQ [26]. Kafka Streams excels in scalability and fault tolerance but might not fit resource-constrained environments due to its complex setup. Azure Event Grid shines within the Azure ecosystem, providing managed autoscaling. RabbitMQ, known for its adaptability, supports varied protocols but can require nuanced configuration.

3.2. Key Components in the Proposed EDA-MQTT Framework

The proposed framework using EDA and MQTT broker for smart transportation comprises five key components, as shown in Figure 1.

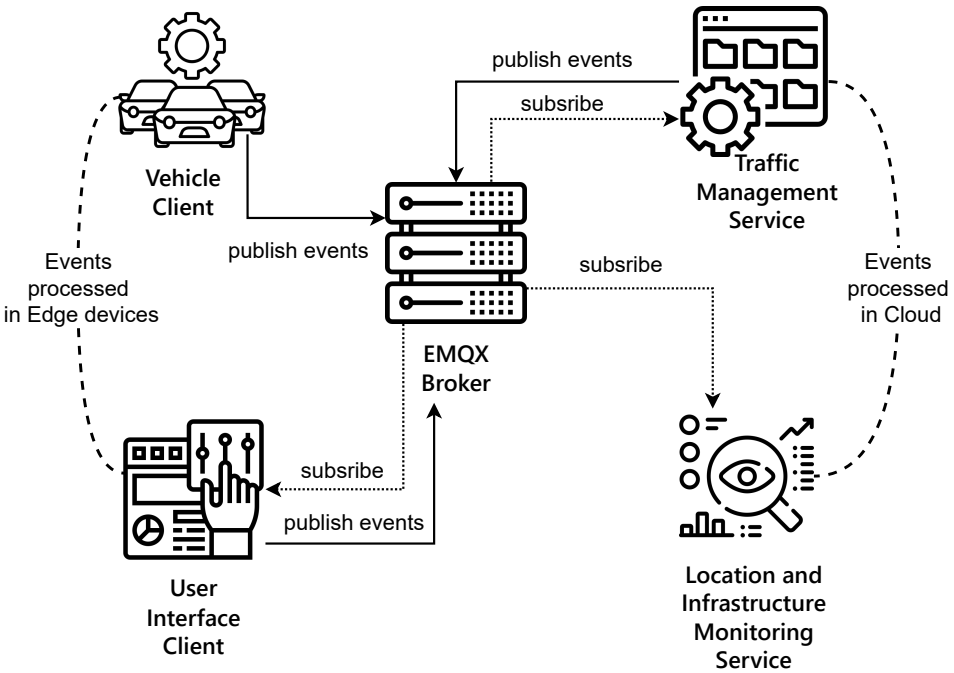

Figure 1. The components in the proposed framework.

- EMQX Broker: The core messaging hub facilitating event-driven communication. It receives events from various clients (user interface client, vehicle client, traffic management service, location and infrastructure monitoring service) and is responsible for accurately forwarding these messages to other clients that have subscribed to them.
- User Interface Client (edge computing component): Serves as the event producer and event consumer. It represents the interface for all end-users, from pedestrians to vehicle operators. It publishes events like traffic congestion reports and listens to updates like route optimizations or transportation schedules.

- Vehicle Client (edge computing component): Mainly serves as the event producer. It caters to all transportation modes, from cars to buses in the edge. This component handles transport-specific events like maintenance alerts, vehicle statuses, and location data.
- Traffic Management Service (hybrid cloud and edge computing component): This service functions as a critical decision-making engine within our framework, operating both at the edge and in the cloud to leverage the strengths of each environment.
  - Edge Deployment: At the edge, the Traffic Management Service focuses on real-time data processing and swift decision making. This proximity to the data sources allows for immediate responses to dynamic traffic conditions, such as adjusting traffic signals to alleviate congestion or responding to unexpected incidents like accidents or road closures. The edge-based component ensures minimal latency and maximizes the responsiveness of the traffic system.
  - Cloud Deployment: In the cloud, the Traffic Management Service undertakes a more comprehensive role. Utilizing the cloud's extensive computational power and vast data storage capabilities, it conducts complex analyses of traffic patterns, predictions of future trends, and development of long-term traffic strategies. The cloud-based service also performs validation and verification of the decisions made at the edge, ensuring overall system accuracy and reliability.
- Location and Infrastructure Monitoring Service (cloud computing component): This service continuously monitors events published by edge servers, facilitating a global analysis of the accumulated data. It rapidly responds to and computes related services, integrating insights from across the network. Additionally, this service is responsible for storing data, ensuring that valuable information is retained for long-term analysis and strategic planning.

Compared to other architectural paradigms, our EDA stands out for its event-focused and asynchronous nature. It contrasts with synchronous patterns like model-view-controller (MVC) and modularity-emphasizing microservices, which can sometimes involve synchronous calls. Our architecture also shares parallels with the event sourcing pattern but emphasizes reactive event handling rather than mere event logging.

Capitalizing on EDA's inherent strengths and MQTT protocol, our architecture strives for scalability, instantaneous responsiveness, and fault resilience, making it suitable for the ever-evolving needs of smart transportation systems.

3.3. MQTT for Event Transport in Edge Computing

Edge computing necessitates efficient communication protocols for real-time data processing, resource-constrained environments, and seamless device/sensor interactions. Choosing the right protocol significantly impacts component interaction, latency, bandwidth utilization, scalability, and security—all crucial factors in edge computing.

Given these demands, MQTT (Message Queuing Telemetry Transport) [27] emerges as a prime candidate. Its lightweight footprint aligns well with the resource limitations of edge devices. Built on a publish–subscribe model, MQTT inherently facilitates event-driven communication, where events are transported through clients subscribing and publishing messages to the broker. Moreover, MQTT offers various quality of service (QoS) levels, ensuring reliable message delivery. This combination of features makes MQTT a well-balanced choice for edge computing requirements.

The following discussion details the rationale behind selecting MQTT for our proposed architecture.

3.3.1. Protocol Comparison for Edge Computing

CoAP (Constrained Application Protocol), AMQP (Advanced Message Queuing Protocol), MQTT, and HTTP (HyperText Transfer Protocol) are popular messaging protocols for IoT and edge computing [28]. While HTTP boasts widespread support and robustness, its resource demands outweigh its benefits in resource-constrained edge environments. Both

CoAP and MQTT excel in low-bandwidth and resource-constrained settings, even running on 8-bit microcontrollers with minimal memory. However, MQTT provides superior throughput and more reliable data delivery options through its QoS levels across low- and high-traffic scenarios. In contrast, AMQP, although adept at complex messaging patterns, falls short in edge computing due to its larger header size and increased complexity [29].

3.3.2. Leveraging MQTT 5.0 for Scalability and Feature Enhancement

The introduction of MQTT 5.0 significantly strengthens its scalability and caters to both large-scale systems and small clients. Features like shared subscriptions and message expiry enhance message management and distribution in large deployments [27]. Session and message expiry intervals offer greater control over session states and message lifetimes, optimizing resource usage, especially in resource-constrained environments. Additionally, MQTT 5.0 introduces new message properties like content type, correlation data, and user properties, enriching the contextual information available for complex data processing and real-time decision making in edge computing systems.

3.3.3. Elixir and MQTT: A Symbiotic Synergy for Edge Computing

Compared to other languages, Elixir's tight integration with MQTT offers robust concurrency processing capabilities. Elixir's lightweight process model and extensive use of asynchronous messaging enable efficient multicore resource utilization, leveraging parallel computing power. Each MQTT connection and session can map to an individual Elixir process, transparently allocated across CPU cores by the scheduler. This one-to-one mapping allows Elixir to handle massive concurrent MQTT connections with superior performance, showcasing its parallel processing capabilities. Studies have demonstrated that Elixir MQTT brokers can support millions of connections with significantly lower latency than other languages.

Furthermore, Elixir's distributed capabilities align seamlessly with MQTT clustering. Through named process registration, Elixir nodes can easily collaborate to construct a geographically distributed, logically unified large-scale MQTT cluster. This cluster automatically load-balances and provides redundancy for fault tolerance, effortlessly managing vast numbers of users and messages.

Finally, Elixir's functional characteristics offer succinct pattern matching capabilities for handling MQTT events, reducing code complexity. Mature MQTT client/server libraries in the Elixir ecosystem expedite development.

In conclusion, Elixir's tight integration with MQTT establishes a powerful programming framework, synergistically combining their strengths to achieve optimal performance and functionality in edge computing applications.

4. Two Case Studies of the Proposed Framework

This section presents two case studies demonstrating the efficacy and versatility of the proposed framework within the domain of smart transportation.

4.1. Traffic Light System for Smart Transportation

4.1.1. Motivation and Approach

Traditional schedule-driven traffic light systems often struggle to adapt to dynamic traffic conditions, leading to unnecessary delays and congestion. To address these limitations, we propose an event-driven architecture (EDA) for adaptive traffic light control within the context of smart transportation. This innovative approach leverages real-time data from diverse sources to dynamically adjust signal timings, potentially minimizing delays and promoting smoother traffic flow compared to fixed-schedule systems [30].

Our proposed system builds upon existing advanced traffic control systems (ATCSs) by incorporating an event-driven paradigm. This enables seamless communication and response between various system components acting as both event producers and consumers. Real-time events such as pedestrian crossings, accidents, and congestion can be

efficiently communicated and acted upon, empowering the system with rapid adaptability to changing traffic conditions. Elixir's inherent scalability and concurrency, as discussed in previous sections, further contribute to the system's responsiveness and effectiveness in handling dynamic traffic patterns.

In addressing the specific scheduling challenges of traffic light signals, our approach incorporates the Oldest Arrival First (OAF) algorithm [31]. The OAF algorithm, known for its efficiency in vehicular traffic scheduling, utilizes real-time position and speed data of individual vehicles. By focusing on isolated traffic intersections, the OAF algorithm aims to minimize delays, enhancing the overall fluidity of traffic movement. By integrating real-time data received from sensors into this algorithm, we can dynamically adjust the scheduling of traffic lights based on real-time analysis. The output of the OAF algorithm, combined with our EDA's responsive policies, allows us to effectively address constantly changing traffic events.

The OAF algorithm's unique capability to process per-vehicle data enables us to dynamically adjust traffic signals in response to immediate traffic conditions. The synergy between the OAF algorithm and our EDA forms the foundation of our proposed smart transportation system.

4.1.2. System Architecture and Implementation

We demonstrate the effectiveness of our approach using a busy crossroads as a testbed, as shown in Figure 2. Sensors strategically placed at the intersection act as event producers, continuously capturing data on vehicle presence, speed, and pedestrian movements. These data points are then transmitted as event messages to roadside units (RSUs) acting as event consumers. The RSUs handle real-time decision making and adjustments based on the received information.

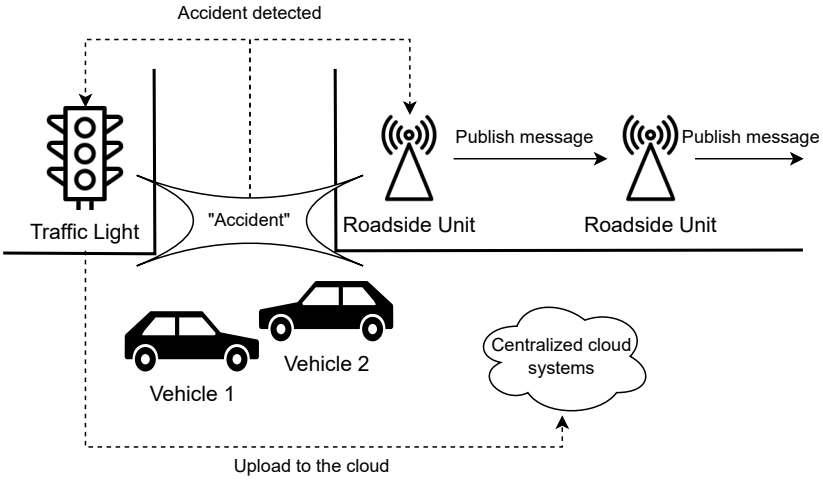

Figure 2. Typical use case of a traffic light system.

The core functionality revolves around event propagation and response:

- When congestion or an accident is detected, the closest RSU publishes a unique event message to the MQTT broker.
- This triggers downstream RSUs on the same street to receive the message and dynamically calculate new route plans for affected vehicles, potentially miles away from the initial event.
- The edge server, running on Elixir, executes immediate control actions based on the updated route plans.
- Centralized cloud systems oversee the broader traffic network and initiate larger adjustments when necessary.

Elixir plays a crucial role in ensuring data integrity and consistency throughout this process. Its functional programming paradigm, featuring immutable data structures,

safeguards data from inadvertent modifications. For instance, an intersection sensor's real-time vehicle density data are represented as an immutable structure shared with two processes: one analyzing city-wide traffic flow and another managing the specific intersection. If the centralized analysis process predicts modifications elsewhere based on this shared data, it does not alter the original structure. This ensures that the intersection control process operates with unaltered data, maintaining consistent and accurate signal timing adjustments based on real-time conditions.

4.1.3. Beyond Traffic Flow Optimization

The benefits of this system extend beyond optimized traffic flow and reduced delays. By minimizing stop-and-go driving, a major contributor to urban vehicle emissions, the system indirectly contributes to a lower carbon footprint for smart transportation systems [32]. This aligns with sustainability goals and leads to improved air quality for urban residents.

4.2. Cab Dispatch System

This subsection presents our second case study: an event-driven cab dispatch system designed for edge computing environments. Such environments demand systems capable of efficiently handling diverse and bursty data streams, effectively utilizing multicore architectures, and dynamically adapting to evolving sensor networks. Our prototype system tackles these challenges and showcases the suitability of Elixir for edge computing applications through its robust concurrency and real-time data processing capabilities.

Figure 3 provides an overview of the system architecture. The detailed implementation aspects of this architecture are discussed in the subsequent section. User-facing passenger and driver applications act as event producers. Passengers initiate ride requests, while drivers simulate location updates, both publishing lightweight messages to a central MQTT broker. A dedicated Elixir-based dispatch service acts as an event consumer, subscribing to these messages and dynamically assigning drivers to passengers based on real-time location and predefined rules.

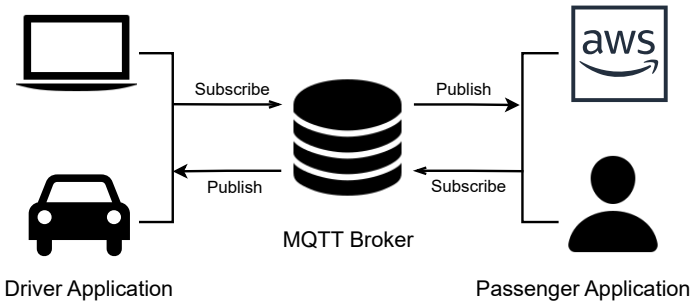

Figure 3. Utilization of the publish/subscribe model in the cab dispatch system.

Elixir's strengths excel in this context, enabling the system to meet the demands of edge computing:

- Lightweight process model: Efficiently manages high concurrency arising from numerous passengers and drivers, eliminating performance bottlenecks.
- Asynchronous messaging: Facilitates real-time responses, ensuring a quick and efficient dispatching experience with minimal latency.

By leveraging these key advantages, our prototype demonstrates the effectiveness of Elixir in latency-sensitive edge computing applications. It paves the way for wider adoption of Elixir in similar scenarios, particularly those requiring robust real-time processing and reliable service delivery.

5. Details of Prototype System

5.1. Event-Driven Design and Components

Figure 4 delves deeper into the event-driven design of the cab dispatch system, highlighting its core components and interactions. In this prototype system, events, event producers, and event channels are implemented as follows.

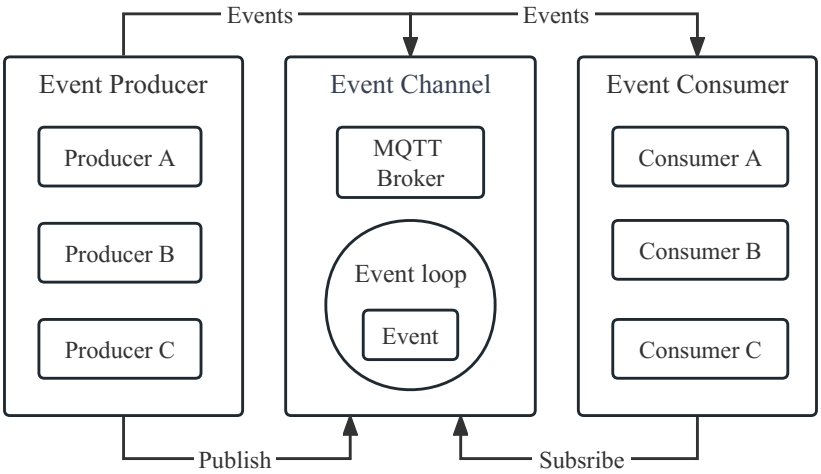

Figure 4. Workflow of the event-driven architecture for the smart transportation system.

5.1.1. Events

The system operates on a foundation of defined events, representing significant occurrences within the system's lifecycle. Core event types include the following:

- CabRequested: Initiated by a passenger application, signifying a ride request with details like passenger ID and destination.
- CabRequestAccepted: Emitted by a driver upon accepting a ride request, confirming their intent to fulfill the request.
- CabArrived: Indicates the driver's arrival at the passenger's pickup location.
- TripStarted: Marks the commencement of the passenger's journey with the chosen driver.
- TripEnded: Signifies the completion of the trip upon reaching the desired destination.

Additional event types can be readily incorporated to cater to future needs and system enhancements. These events are structured data payloads published to the MQTT broker, facilitating efficient and decoupled communication. Upon receiving an event, the dispatch service transitions the corresponding request through its state machine, moving it from pending to assigned, and, subsequently, through other relevant states. For instance, a CabRequest event triggers the transition from pending to assigned, while a TripEnded event marks the final state. This explicit state management allows for clear observation and control of the system's evolution.

5.1.2. Event Producers

Proactive entities within the system act as event producers, encapsulating complex backend details and emitting only meaningful events. Different entities can assume the role of producer depending on the context. In our case, the passenger and driver applications serve as primary producers:

- Passenger application: Publishes CabRequested events upon initiating ride requests.
- Driver application: Publishes CabRequestAccepted events when accepting ride requests and periodically generates LocationUpdate events to broadcast their current location.

This producer-driven approach enables consistent event exposure, simplifying the integration of new data sources and promoting system flexibility.

5.1.3. Event Channels

The MQTT broker serves as the system's central event channel, providing reliable and persistent message transmission between producers and consumers. It acts as a message broker, actively listening for incoming connections, requests, and messages. When a producer publishes an event, the broker filters, validates, and queues it for replayable delivery, ensuring message integrity and resilience. This asynchronous dispatching facilitates decoupled services and enhances the overall effectiveness of the event-driven architecture. The broker's event loop architecture efficiently distributes millions of events per second across geographically distributed edge devices, enabling real-time responsiveness and scalability.

5.2. Cab Dispatch Scenario: A Sequence of Events

The cab dispatch scenario unfolds through a series of orchestrated interactions between key system components: the passenger application, MQTT broker, dispatch service, and driver application. Figure 5 visualizes these interactions using a UML sequence diagram, highlighting the chronological flow of events:

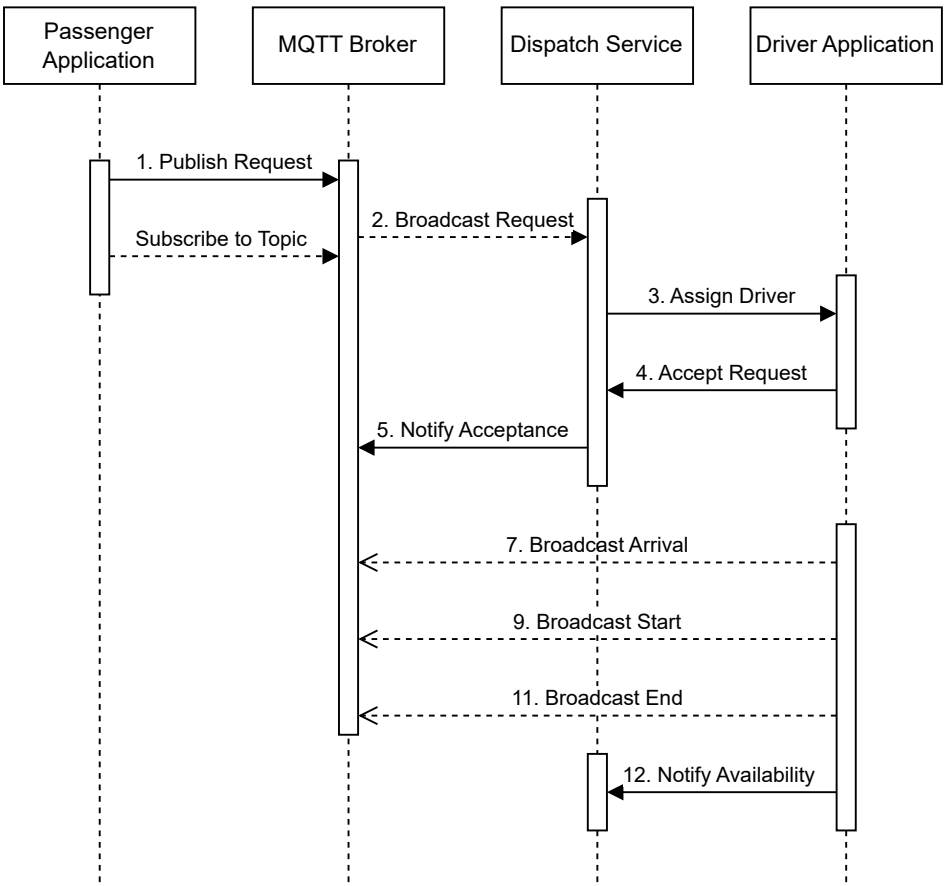

Figure 5. The sequence diagram of the cab dispatch system.

1. Cab Requested: Initiated by the passenger application, this event signifies a ride request and includes details like passenger ID and desired destination. The passenger application publishes this event to the MQTT broker and subscribes to the topic for receiving cab acceptance notifications.
2. Cab Request Broadcasted: Upon receiving the CabRequested event, the MQTT broker acts as a message intermediary, broadcasting it to all subscribed entities, including the dispatch service and driver applications.
3. Driver Assigned: The dispatch service receives the broadcasted request and, applying predefined rules like driver proximity and availability, assigns the request to the most suitable driver.

4. Cab Request Accepted: The assigned driver acknowledges the request by publishing a CabRequestAccepted event to the MQTT broker.
5. Acceptance Notification Broadcasted: Similar to the request, the broker relays the acceptance notification to all subscribers, including the passenger application, confirming the assigned driver.
6. Cab Arrived: The driver application simulates the driver's journey to the passenger's location and, upon arrival, publishes a CabArrived event to notify the system.
7. Arrival Notification Broadcasted: The broker forwards the arrival notification to subscribed entities, informing the passenger that the driver has arrived at the pickup point.
8. Trip Started: Once the passenger boards the cab, the driver application publishes a TripStarted event to mark the commencement of the trip.
9. Trip Start Notification Broadcasted: The broker disseminates the start notification, informing all subscribers, including the passenger application, that the trip has begun.
10. Trip Ended: Upon reaching the destination, the driver application publishes a TripEnded event to signify the completion of the trip.
11. Trip End Notification Broadcasted: The broker broadcasts the final notification to all subscribers, informing them of the trip's completion.
12. Cab Availability Notification: The driver application updates its status to available by notifying the dispatch service, allowing it to assign subsequent ride requests.

This sequential flow exemplifies the event-driven nature of the system, where individual events trigger specific actions and state transitions, orchestrating the entire cab dispatch process.

5.3. Basic Components

5.3.1. Driver Application

The Driver application, designed for edge devices, continuously updates the driver's location to simulate real-time movement. In the experimental evaluation described in the next section, we achieve this by generating random geographical coordinates at regular intervals. The application comprises two primary modules:

- Location Update Module: Generates random location updates for each driver, mimicking continuous position changes through periodic updates. Listing 1 demonstrates a code snippet showcasing this periodic location update functionality.
- MQTT Publishing Module: Publishes location updates as JSON-encoded messages to the MQTT broker under the driver's unique topic. Additionally, it synchronizes all simulated taxis' statuses and publishes pickup requests when drivers become available.

Listing 1. Updating the driver's current location in real time.

```
def update_location(driver_id) do
  :timer.sleep(1000)

  location = generate_random_location()

  :ok = publish_location(driver_id, location)

  update_location(driver_id)  # Keep the loop going
end

defp generate_random_location() do
  %{latitude: rand(0..90), longitude: rand(0..180)}
end

defp publish_location(driver_id, location) do
  event = %{driver_id: driver_id, location: location, timestamp:
      DateTime.utc_now()}
```

```
17
18    event_binary = Jason.encode!(event)
19
20    :ok = :emqtt.publish(conn, "driver/#{driver_id}/location",
          event_binary)
21  end
```

The Elixir application concurrently manages hundreds of taxi objects, including their real-time location and state changes, replicating an actual taxi dispatch scenario. This application runs autonomously, simulating multiple drivers updating and publishing their locations indefinitely, demonstrating Elixir's robustness and suitability for long-running edge computing processes.

5.3.2. Passenger Application

Hosted on the AWS cloud, the passenger application allows passengers to initiate ride requests. It serves as the entry point for submitting requests, specifying details like passenger ID and destination. After submitting a request, the application establishes an MQTT broker connection and subscribes to dedicated topics based on the request ID.

Listing 2 illustrates this subscription using the :emqtt.subscribe/2 function, focusing on topics like events/cab_requested/#passenger_id, Here, #passenger_id is replaced with the actual ID of the passenger, ensuring that the application only receives updates relevant to its specific requests. This ensures the application receives only updates relevant to its specific request. It then enters a listening state, waiting for incoming MQTT messages that provide real-time ride status updates. These messages are processed to inform the passenger about their ride's current status.

Listing 2. Request status monitoring.

```
1  {:ok, conn} = :emqtt.start_link([clientid: "PassengerApp",
       clean_start: false])
2
3  :ok = :emqtt.subscribe(conn,
       "events/cab_requested/#{passenger_id}")
4
5  receive do
6    {:publish, publish} ->
7      IO.inspect(publish)
8  end
```

5.3.3. Dispatch Service

The dispatch service matches passenger requests available taxis based on proximity. It subscribes to MQTT messages containing passenger requests and, upon receiving one, spawns lightweight Elixir processes to concurrently calculate proximity for all idle taxis. This concurrent approach significantly reduces matching computation time. Additionally, we plan to implement load-balancing algorithms in the future.

The dispatch module publishes the assignment result via MQTT to the assigned driver's topic. If no drivers are available, it sends a notification to the passenger about the failed assignment. Leveraging Elixir's efficient concurrency and distributed communication primitives, this module enables fast and reliable order dispatching.

6. Evaluation

Our evaluation methodology is designed to assess the performance of the cab dispatch system within an edge computing paradigm, focusing on several critical metrics: response time latency, concurrency handling under high load conditions, fault tolerance, and multicore processing efficiency. Recognizing the complexity and variability of real-world IoT hardware environments [33], our current testing uses a purely software-based

environment, and future work will include detailed hardware simulations to enhance the practical applicability and robustness of our findings in diverse IoT contexts. Additionally, we also conducted experiments under simulated constrained network conditions to mimic potential real-life challenges. The testbed for our experiments is a MacBook Pro equipped with an Apple M1 chip, featuring 8 cores (4 performance and 4 efficiency cores), and 8 GB of RAM, running macOS Monterey version 12.6. Our test programs were developed in Elixir version 1.14.3, compiled with Erlang/OTP 25. The EMQX MQTT broker was deployed within a Docker container on macOS, and the MQTT clients were operational on Ubuntu VMs, version 22.04.1.

6.1. Latency Testing

We initially focused on response time, defined as the interval between dispatch request initiation by the driver application and its receipt by the passenger application. More specifically, the driver application sends the location to the broker, and the passenger application subscribing to the same topic in the broker will process the location message from the driver application. This test scrutinized system responsiveness, including Phoenix LiveView's hot code loading capabilities. To mimic real-world conditions, the Locust script's "wait_time" parameter controlled the dispatching of real-time location updates at 10 to 30 s intervals. This parameter means that every virtual driver will send the real-time location in every 10 to 30 s.

Further iterations of the test varied the number of virtual drivers and their spawn rate to observe the system's behavior under different loads. As depicted in Figure 6, we executed five sets of tests with varying user counts and spawn rates. The term "spawn rate" here describes the velocity of virtual user generation per second during the test. For instance, in the first group, with a spawn rate of 2 and a target of 100 users, the system incrementally added two users per second until it reached the full count. The results were promising: the median latency stayed below 5 ms, showcasing efficient data processing for the majority of transactions. At the 80th percentile, even with 2000 users, the latency only peaked around 8 ms, indicating that 80% of the transactions were processed within 8 ms. Such performance underscores the framework's aptness for edge computing scenarios, where swift response times are critical.

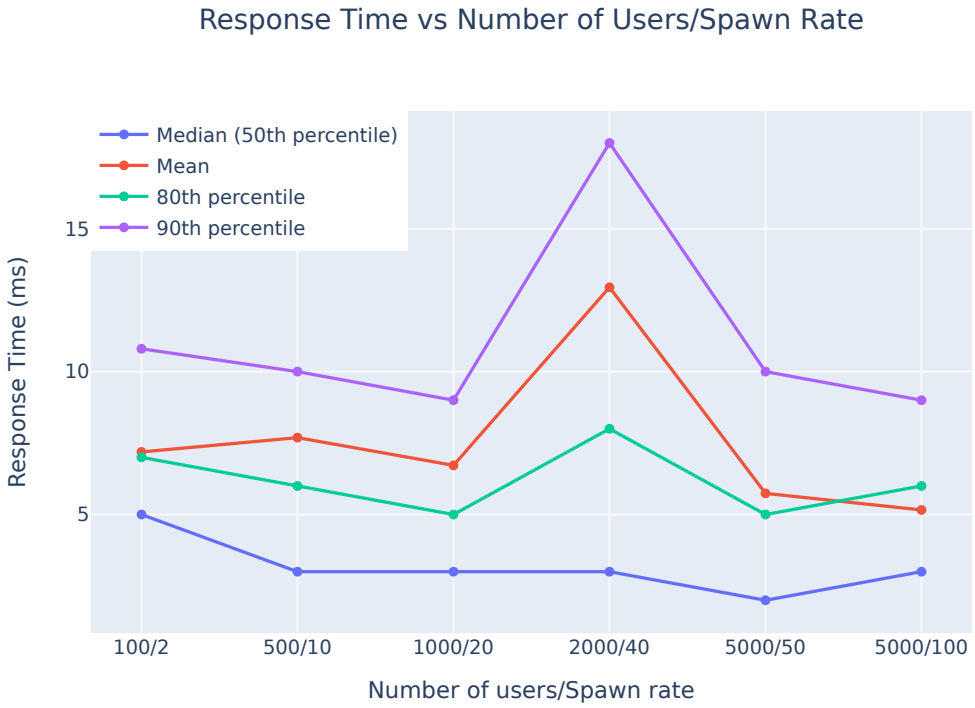

Figure 6. Latency comparison of different groups.

Elixir's functional programming paradigm and the Erlang VM's robust concurrency model contribute to this consistent performance under complex, concurrent processing scenarios. These features equip the system for the real-time demands of edge computing.

Furthermore, to present a more vivid and direct representation of our system's performance, we conducted uninterrupted tests with 2000 and 10,000 users, respectively. Key segments of these tests are illustrated in Figures 7 and 8. The plots predominantly show that the majority of response times are maintained below 8 milliseconds. Notably, while occasional latency spikes were observed, the system leveraged Elixir's fault tolerance capabilities to quickly return to optimal latency ranges under 8 milliseconds. This demonstrates the resilience and stability of our system even under fluctuating conditions.

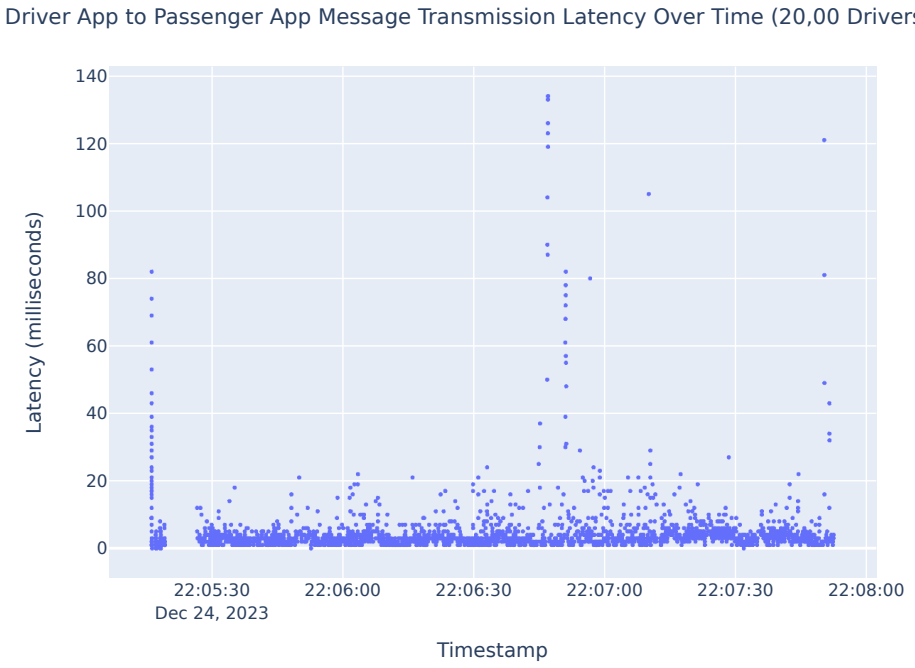

Figure 7. Latency from passenger app to driver app of 2000 users group.

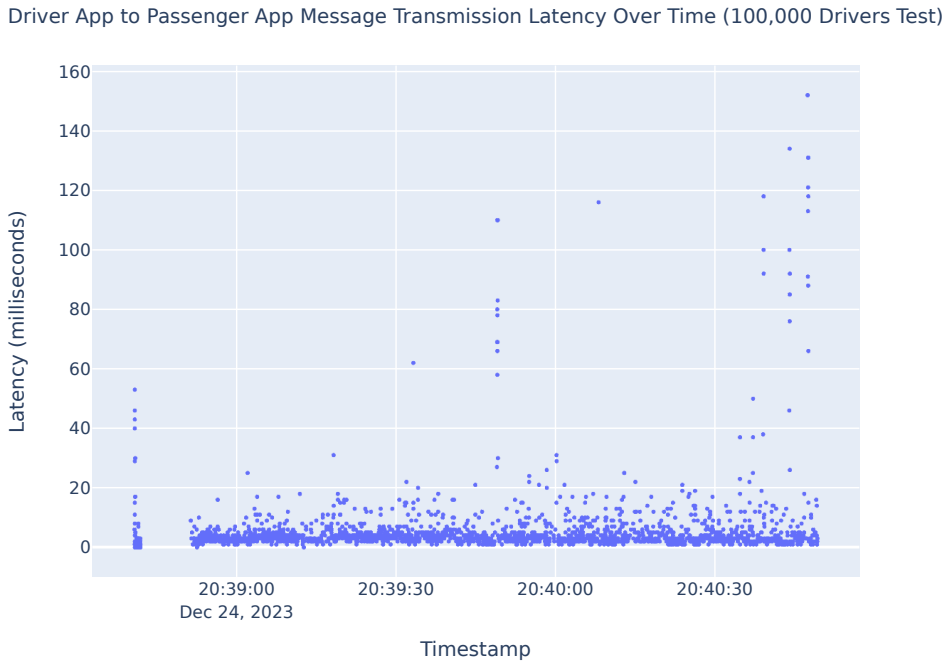

Figure 8. Latency from passenger app to driver app of 10,000 users group.

In scenarios simulating extreme conditions with up to 10,000 users, the system still sustained its performance level. Even with the increased load, the scatterings of peak latency did not exceed 160 ms. This resilience under high user volume is particularly significant, highlighting the system's compatibility with smart transportation applications that demand low latency and high-density user environments.

Latency Comparison with EdgeX Foundry

EdgeX Foundry is a well-known, highly flexible, and scalable open-source edge computing platform that facilitates interoperability between devices and applications at the IoT Edge [34]. We conducted stress tests on the EdgeX Foundry platform under the same testing conditions. Specifically, we configured our test environment on an Ubuntu virtual machine and adjusted the conditions described in Section 6.1. Using EdgeX's built-in device-virtual service, we simulated a cab client and a passenger client, and we conducted the stress tests by using the wrk tool. The tests had the following results: with 8 threads and 50 connections, the average latency was 15.77 ms, with a latency distribution of 75% at 19.04 ms and 90% at 23.52 ms. In comparison, stress tests conducted on our framework using locust demonstrated a lowest average latency of 5 ms, with a 90% latency distribution at 9 ms. This comparative testing showcases the low-latency advantage of our framework. Further detailed comparative research with other frameworks will be conducted in our future work.

6.2. Throughput Testing and Resource Utilization under Load

The next evaluation focused on throughput, analyzing concurrency handling capability under load. The Locust script's "wait_time" was set to 2–8 s to simulate peak traffic conditions, alongside increasing user numbers and spawn rates up to 40,000 users and 5000 spawn rate. Requests per second (RPS) serves as a crucial metric reflecting the system's ability to process incoming requests. Figure 9 illustrates the system's consistent RPS performance across various stress conditions. The results confirm robust performance, efficiently handling high request volumes during peak traffic periods.

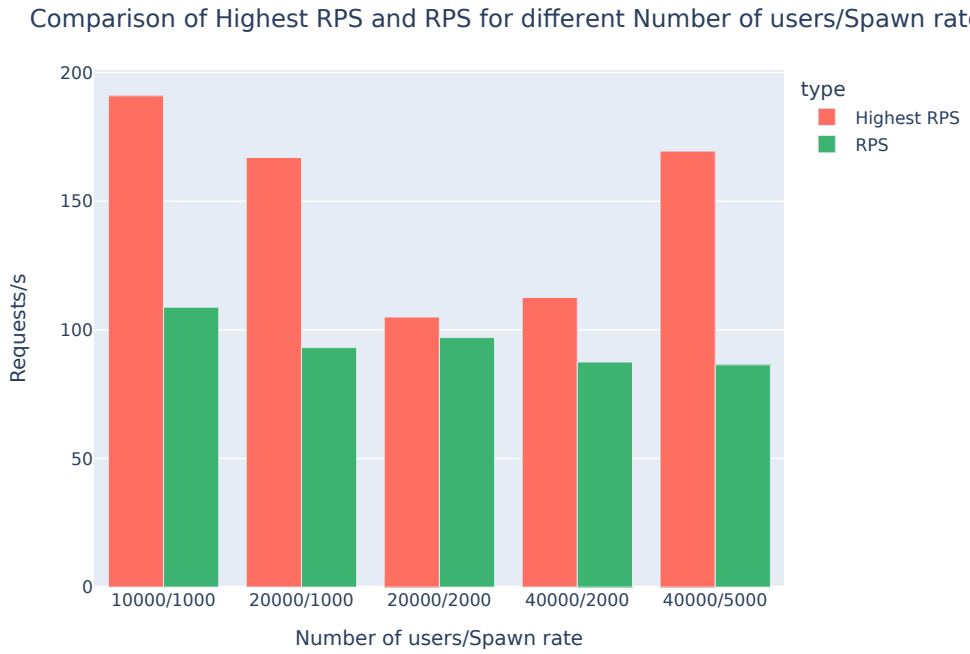

Figure 9. Throughput testing in different sets of groups.

Additionally, resource utilization, defined as the ratio of the actual time spent by resources executing the workload, is another critical metric in the edge computing paradigm [35]. In the throughput testing under load, we also monitored the CPU uti-

lization to assess the overall resource utilization. We focused specifically on the 40,000 user and 5000 spawn rate scenario, and observed a peak CPU utilization of 58%. This figure indicates a balanced resource usage. While lower utilization suggests the availability of additional resources, higher average CPU utilization does not necessarily equate to more efficient orchestration [36]. Indeed, high levels of CPU utilization might indicate load accumulation on other resources due to inefficient load or resource management [37]. The relatively modest resource utilization observed in our tests could be attributed to Elixir's lightweight process model. However, it also highlights opportunities for further optimization of resource usage in future work.

6.3. Constrained Network Conditions Testing

In real-world edge network environments, a variety of factors can contribute to unstable network conditions. Examples in edge networks include varying signal strengths due to geographical locations, network congestion caused by high user density, and intermittent connectivity issues in remote areas. Testing system performance under such unstable network conditions is crucial for improving our framework in edge computing scenarios.

We set three groups of contained network conditions and evaluated them as shown in Figure 10.

Metrics	# Requests	# Fails	Average RPS	Mean	90th percentage
Constraned 1	8184	0	28	114.19	119
Constraned 2	4575	0	20.7	121.32	165
Constraned_3	6358	0	22.6	29.92	20

Figure 10. Different constrained network conditions testing.

- Constrained_1: This scenario involved a network condition with a fixed delay of 100 ms and a packet loss rate of 1%. Under these conditions, the system managed to handle 8184 requests without any failures, maintaining an average RPS of 28.0. The mean latency was recorded at 114.19 ms, with the 90th percentile latency reaching 119.0 ms.
- Constrained_2: This scenario set a base latency of 100 ms with a random fluctuation of up to ±20 ms. The system processed 4575 requests and exhibited an average RPS of 20.7. The mean latency increased slightly to 121.32 ms, and the 90th percentile latency rose to 165.0 ms, reflecting the impact of variable delay conditions.
- Constrained_3: This scenario tested the system's response to a higher packet loss rate of 5%. Despite this challenge, the system processed 6358 requests and achieved an average RPS of 22.6. Without extra latency added in this test, the mean latency in this scenario was significantly lower, at 29.92 ms, and the 90th percentile latency was 20.0 ms.

For comparison, under normal conditions without any imposed network constraints, the average latency of the system was around 8 ms. The observed latency showed an accordingly additional increase over the standard measurements, demonstrating a linear response to the network delay introduced.

6.4. Fault Tolerance Testing

In fault tolerance testing, we designed two primary experimental scenarios to assess the system's performance on fault tolerance: interruptions of the MQTT Broker service and terminations of Elixir processes. The system's response in these scenarios reflects its recovery capabilities and robustness.

- MQTT Broker Service Interruption: In this experiment, we demonstrate the system's resilience when a critical communication component fails. With the MQTT broker positioned at the network edge, maintaining the integrity of complex event transmissions is crucial, especially when encountering unforeseen incidents such as power

outages. Equally important is the ability of MQTT clients to reconnect and quickly return to low-latency operations. As shown in the logs and the accompanying figure, we simulated the latency variations during a broker power outage and subsequent restart. Here, latency refers to the delay experienced by the passenger application in reconnecting to the restarted broker. The log data, with initial latencies exceeding 3000 ms back to the average condition of 10 ms, indicate that the reconnected passenger application rapidly processed messages sent during the outage. When the broker is powered down, the connection between the broker and clients breaks down immediately, the message sent by the driver application will be stored in the broker immediately, and the passenger will automatically keep trying to reconnect and listen to the same topic. When the MQTT broker is powered again, the system will process the omitted events first then quickly return back to normal performance. The system swiftly transitioned from high latency back to low latency, demonstrating rapid recovery capabilities.

- Elixir Process Interruption:
  Before the details of the experimental setup, it is necessary to briefly explain the language logic of Elixir that underpins its fault tolerance.
  In BEAM-based systems, resilience is fundamentally rooted in the complete isolation and independence of each process. This isolation is a key aspect of Elixir's fault tolerance, ensuring that a crash in one process does not affect others. BEAM processes are lightweight, concurrent entities managed by the virtual machine (VM), which schedules their execution. Typically, BEAM employs as many schedulers as there are available CPU cores, with each scheduler operating in its own thread, while the entire VM runs within a single OS process [38]. Figure 11 shows a simplified version of the real experiment environment. Moreover, each process can maintain its state and communicate with other processes to manipulate or access this state. In Elixir, data immutability is a core principle. Processes act as containers for this data, preserving immutable structures over time, sometimes indefinitely.
  In addition to its inherent resilience, Elixir further enhances fault tolerance through the implementation of specialized supervisor processes. These supervisor processes are solely responsible for supervision processes, which are called child processes. Whenever a child process terminates, the supervisor promptly initiates a replacement, utilizing various strategies to manage these processes. This approach effectively reduces the cascading failures and the frequency of restarts that might be triggered by linked process crashes.
  Our experimental setup within the passenger application is depicted in Figure 11, which includes a primary supervisor executing the `rest_for_one` strategy. This strategy ensures that when a child process crashes, the supervisor terminates all subsequently listed processes in the child specification. This is particularly crucial for Process_1, which gathers other system components and needs to maintain consistent state updates to avoid data conflicts. Both the SystemSupervisor and ServerSupervisor employ the `one_for_one` strategy, whereby the crashing and restarting of their supervised processes do not impact the operation of others.
  The experimental results under the supervisor tree are presented in Table 1. Differing from the random process terminations of chaos monkey experiments, we deliberately terminated processes to observe the anticipated restart/no-restart behaviors. As shown in the table, the outcomes matched our expectations: terminating Process_1 led to the restart of all processes; terminating Process_2 did not trigger any restarts; and since Process_3 and Process_4 are linked, the crash of either affected the other. In summary, Elixir's process and supervisor mechanisms flawlessly uphold its fault tolerance capabilities. Within the edge computing paradigm, these features can offer vast prospects for smart transportation applications.

Table 1. Testing of processes to be killed and the existence/nonexistence.

Process \ Killed Process	Process_1	Process_2	Process_3	Process_4
ine Process_1		✓	✓	✓
ine Process_2	✗		✓	✓
ine Process_3	✗	✓		✓
ine Process_4	✗	✓	✗	

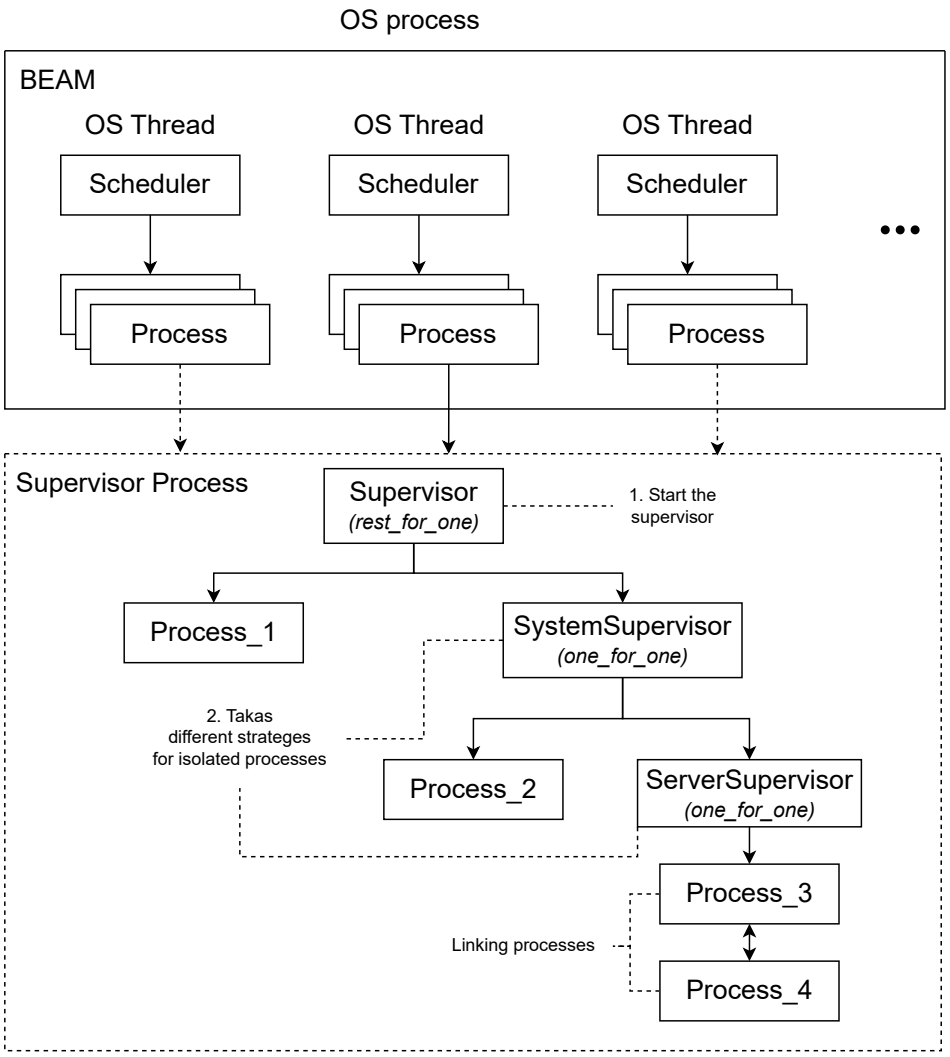

Figure 11. BEAM works as a single OS process and the processes under the supervisor tree.

6.5. Multicore Programming

Listing 3 presents a simulation involving 500 concurrent events, each calculating the Euclidean distance from the origin to a random point. By invoking :erlang.system_info, we determined the number of schedulers used by the Erlang runtime system (typically one scheduler per CPU core). Not only for this individual experiment, but all the above testings were also monitored to use multicore during the process. Our results confirmed

that the Erlang runtime utilized all eight cores of the Apple M1 chip, demonstrating Elixir's effectiveness in leveraging multicore architecture.

Listing 3. Elixir multicore programing example.

```
defmodule ConcurrencyTest do
  def distance_to_origin(x, y), do: :math.sqrt(x * x + y * y)

  def test_concurrency(num_tasks) do
    tasks = 1..num_tasks
    |> Enum.map(fn _ ->
        Task.async(
          fn ->
            x = :rand.uniform(1000)
            y = :rand.uniform(1000)
            distance_to_origin(x, y)
          end) end)
    |> Enum.map(&Task.await/1)

    IO.puts("Schedulers:
        #{:erlang.system_info
        (:schedulers_online)}")
  end
end

ConcurrencyTest.test_concurrency(500)

IO.inspect(:erlang.system_info
    (:schedulers_online))
```

6.6. Summary

The evaluations, encompassing both response time and throughput under high load, demonstrate the system's reliability and scalability. In an edge computing environment, the cab dispatch system exhibits both agile response and sturdy concurrency management, making it well suited for edge-centric solutions.

7. Concluding Remarks

This paper proposed an edge computing architecture tailored for real-time, distributed applications. Our solution leverages Elixir's unique capabilities: its lightweight concurrent processing model for efficient resource utilization, and robust fault tolerance mechanisms inherited from Erlang/OTP for enhanced system resilience. Additionally, we employed the MQTT protocol for asynchronous event transport due to its proven efficiency and reliability in distributed environments.

The evaluation of our framework through a cab dispatch prototype demonstrated its strengths. The prototype achieved low-latency and high-concurrency capabilities, underlining the effectiveness of Elixir's multicore utilization for edge computing scenarios. This successful prototype exemplifies the potency of Elixir in crafting scalable and responsive edge applications.

Our findings offer a valuable contribution to the field by furthering the integration of Elixir and event-driven models within edge computing domains. This paves the way for further scholarly discourse and exploration of Elixir's potential in addressing the evolving demands of real-time distributed systems at the edge.

In this study, we presented two case studies that collectively illustrate the unique advantages of adopting event-driven architecture (EDA) and the Message Queuing Telemetry Transport (MQTT) protocol in smart transportation applications, demonstrating through

testing the irreplaceable benefits of Elixir's integration for system performance enhancement. However, our experiments were conducted in a purely software environment; hence, testing for broader application scenarios remains insufficient. As mentioned in the related research section, we plan to continue framework improvement and testing on hardware systems supporting the BEAM system, such as the Erlang-based GRiSP hardware platform, aiming to propose a more comprehensive and application-enhanced version.

The research into these two case studies has led us to identify the optimal integration of EDA and smart transportation, notably in rapid response to emergency events and in reducing decision-making time during peak periods. At the same time, we have identified other areas requiring further exploration, such as ensuring data protection while maintaining efficient communication. Although Elixir and MQTT 5.0 provide us with high levels of security, further research in this area will also be a part of our future work.

Author Contributions: Conceptualization, Y.L. and S.F.; methodology, Y.L. and S.F.; software, Y.L.; validation, Y.L.; writing—original draft preparation, Y.L.; writing—review and editing, Y.L. and S.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to as the data are part of an ongoing study.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Pan, J.; McElhannon, J. Future edge cloud and edge computing for internet of things applications. *IEEE Internet Things J.* **2017**, *5*, 439–449. [CrossRef]
2. Yu, W.; Liang, F.; He, X.; Hatcher, W.G.; Lu, C.; Lin, J.; Yang, X. A survey on the edge computing for the Internet of Things. *IEEE Access* **2017**, *6*, 6900–6919. [CrossRef]
3. Statista. Internet of Things (IoT) Total Annual Revenue Worldwide from 2020 to 2030. 2021. Available online: https://www.statista.com/statistics/1194709/iot-revenue-worldwide/ (accessed on 15 January 2024).
4. Grand View Research. Edge Computing Market Size, Share & Trends Analysis Report by Component, by Application (Smart Grids, Remote Monitoring), by End Use (Manufacturing, Healthcare), by Region, and Segment Forecasts, 2020–2027. 2021. Available online: https://www.grandviewresearch.com/industry-analysis/edge-computing-market (accessed on 15 January 2024).
5. Li, Y.; Fujita, S. Design of Elixir-Based Edge Server for Responsive IoT Applications. In Proceedings of the 2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW), Himeji, Japan, 21–24 November 2022; pp. 185–191. [CrossRef]
6. Rahmani, A.M.; Babaei, Z.; Souri, A. Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing. *Clust. Comput.* **2021**, *24*, 1347–1360. [CrossRef]
7. Khazael, B.; Asl, M.V.; Malazi, H.T. Geospatial complex event processing in smart city applications. *Simul. Model. Pract. Theory* **2023**, *122*, 102675. [CrossRef]
8. Alvarez, M.G.; Morales, J.; Kraak, M.-J. Integration and exploitation of sensor data in smart cities through event-driven applications. *Sensors* **2019**, *19*, 1372. [CrossRef] [PubMed]
9. Xiao, C.; Chen, N.; Gong, J.; Wang, W.; Hu, C.; Chen, Z. Event-driven distributed information resource-focusing service for emergency response in smart city with cyber-physical infrastructures. *ISPRS Int. J. Geo-Inf.* **2017**, *6*, 251. [CrossRef]
10. Saeik, F.; Avgeris, M.; Spatharakis, D.; Santi, N.; Dechouniotis, D.; Violos, J.; Leivadeas, A.; Athanasopoulos, N.; Mitton, N.; Papavassiliou, S. Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions. *Comput. Netw.* **2021**, *195*, 108177. [CrossRef]
11. Luo, Q.; Hu, S.; Li, C.; Li, G.; Shi, W. Resource scheduling in edge computing: A survey. *IEEE Commun. Surv. Tutor.* **2021**, *23*, 2131–2165. [CrossRef]
12. Dunkel, J.; Fernández, A.; Ortiz, R.; Ossowski, S. Event-driven architecture for decision support in traffic management systems. *Expert Syst. Appl.* **2011**, *38*, 6530–6539. [CrossRef]
13. Lee, W.-H.; Chiu, C.-Y. Design and implementation of a smart traffic signal control system for smart city applications. *Sensors* **2020**, *20*, 508. [CrossRef]
14. Ke, R.; Cui, Z.; Chen, Y.; Zhu, M.; Yang, H.; Wang, Y. Edge computing for real-time near-crash detection for smart transportation applications. *arXiv* **2020**, arXiv:2008.00549.

15. Kopestenski, I.; Van Roy, P. Achlys: Towards a framework for distributed storage and generic computing applications for wireless IoT edge networks with Lasp on GRiSP. In Proceedings of the 2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), Kyoto, Japan, 11–15 March 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 875–881.
16. Kopestenski, I.; Van Roy, P. Erlang as an Enabling Technology for Resilient General-Purpose Applications on Edge IoT Networks. In Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang, Berlin, Germany, 18 August 2019; pp. 1–12.
17. Kalbusch, S.; Verpoten, V.; Van Roy, P. The Hera Framework for Fault-Tolerant Sensor Fusion with Erlang and GRiSP on an IoT Network. In Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Virtual, 26 August 2021; pp. 15–27.
18. Armstrong, J. A History of Erlang. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, San Diego, CA, USA, 9–10 June 2007; pp. 6-1–6-26.
19. Lilis, Y.; Savidis, A. A survey of metaprogramming languages. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–39. [CrossRef]
20. Michelson, B.M. Event-driven architecture overview. *Patricia Seybold Group* **2006**, *2*, 10–1571.
21. Taylor, H.; Yochem, A.; Phillips, L.; Martinez, F. *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*; Addison-Wesley Professional: Boston, MA, USA, 2009.
22. Khriji, S.; Benbelgacem, Y.; Chéour, R.; El Houssaini, D.; Kanoun, O. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *J. Supercomput.* **2022**, 1–28. [CrossRef]
23. Goyal, P.; Mikkilineni, R. Policy-based event-driven services-oriented architecture for cloud services operation & management. In Proceedings of the 2009 IEEE International Conference on Cloud Computing, Bangalore, India, 21–25 September 2009; pp. 135–138.
24. Apache Kafka. Kafka Streams. Available online: https://kafka.apache.org/documentation/streams/ (accessed on 15 January 2024).
25. Microsoft Azure. Azure Event Grid. Available online: https://azure.microsoft.com/en-us/services/event-grid/ (accessed on 15 January 2024).
26. RabbitMQ. RabbitMQ. Available online: https://www.rabbitmq.com/ (accessed on 15 January 2024).
27. *MQTT Version 5.0*, Technical Report; OASIS Standard: Woburn, MA, USA, 2019. Available online: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (accessed on 22 June 2020).
28. Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In Proceedings of the 2017 IEEE International Systems Engineering Symposium (ISSE), Vienna, Austria, 11–13 October 2017; pp. 1–7.
29. Ouakasse, F.; Rakrak, S. A comparative study of MQTT and COAP application layer protocols via. performances evaluation. *J. Eng. Appl. Sci.* **2018**, *13*, 6053–6061.
30. Zhao, Y.; Tian, Z. An overview of the usage of adaptive signal control system in the United States of America. *Appl. Mech. Mater.* **2012**, *178*, 2591–2598. [CrossRef]
31. Pandit, K.; Ghosal, D.; Zhang, H.M.; Chuah, C.-N. Adaptive traffic signal control with vehicular ad hoc networks. *IEEE Trans. Veh. Technol.* **2013**, *62*, 1459–1471. [CrossRef]
32. Zhao, C.; Wang, K.; Dong, X.; Dong, K. Is smart transportation associated with reduced carbon emissions? The case of China. *Energy Econ.* **2022**, *105*, 105715. [CrossRef]
33. Belson, B.; Holdsworth, J.; Xiang, W.; Philippa, B. A survey of asynchronous programming using coroutines in the Internet of Things and embedded systems. *ACM Trans. Embed. Comput. Syst. (TECS)* **2019**, *18*, 1–21. [CrossRef]
34. EdgeX Foundry. n.d. Available online: https://www.edgexfoundry.org/ (accessed on 15 January 2024).
35. Gill, S.S.; Chana, I.; Singh, M.; Buyya, R. CHOPPER: An intelligent QoS-aware autonomic resource management approach for cloud computing. *Clust. Comput.* **2018**, *21*, 1203–1241. [CrossRef]
36. Aslanpour, M.S.; Gill, S.S.; Toosi, A.N. Performance evaluation metrics for cloud, fog and edge computing: A review, taxonomy, benchmarks and standards for future research. *Internet Things* **2020**, *12*, 100273. Available online: https://www.sciencedirect.com/science/article/pii/S2542660520301062 (accessed on 15 January 2024). [CrossRef]
37. Aslanpour, M.S.; Ghobaei-Arani, M.; Toosi, A.N. Auto-scaling web applications in clouds: A cost-aware approach. *J. Netw. Comput. Appl.* **2017**, *95*, 26–41. [CrossRef]
38. Juric, S. *Elixir in Action*; Simon and Schuster: New York, NY, USA, 2019.