ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria - DISI

Corso di Laurea in Ingegneria Informatica

Real-Time IoT Framework for Data Continuity of Digital Shadows

Relatore
Chiar.mo Prof. Paolo Bellavista
Chiar.mo Prof. Luca Foschini

Presentata da
Stefano Spadari

Correlatore
Chiar.mo Prof. Berk Canberk

IN COLLABORATION WITH
Edinburgh Napier University

Sessione Marzo 2026
Anno Accademico 2024/2025

Contents

List of Figures

Chapter 1

Introduction

1.1 Background: Real-Time IoT Systems and Data Pipelines

Over the past decade, the widespread availability of low-cost sensors has enabled the development of Internet of Things (IoT) systems capable of collecting environmental, operational, or structural measurements at high frequencies [1]. Devices deployed in buildings, industrial infrastructures, or urban environments produce continuous data streams that must be delivered to processing and storage services in real time.

Pipeline-based architectures are commonly adopted in modern IoT systems to ensure scalability and structured data processing. In these architectures, sensor data typically flow through acquisition or ingestion components, intermediate processing stages for cleaning, transformation, and integration, and storage services optimized for downstream analytics and visualization [2]. The transformation phase is particularly critical, as raw data may contain errors, duplicates, and missing values that must be handled to guarantee consistency.

Communication between system components is often supported by the MQTT protocol, a lightweight publish/subscribe messaging system designed for resource-constrained devices and low-bandwidth networks [3]. Thanks to its quality-of-service levels and support for persistent sessions, MQTT can provide reliable message delivery even under variable network conditions.

Within this context, continuous data streams are frequently used to maintain virtual representations of physical systems, leading to the concept of *digital shadow*. A digital shadow is a virtual representation of a physical entity that is automatically updated through a one-way data flow from the physical system to the digital one. Unlike a digital twin, it does not imply a bidirectional interaction with the physical asset, but rather reflects its state for monitoring, analysis, and prediction purposes [4]. Typical examples include telemetry systems in motorsport, where data transmitted from a vehicle are analyzed

in real time without directly affecting its operation. Within this context, continuous data streams are frequently used to maintain virtual representations of physical systems, leading to the concept of ***digital shadow***. A digital shadow is a virtual representation of a physical entity that is automatically updated through a one-way data flow from the physical system to the digital one. **Unlike a digital twin**, it does not imply a bidirectional interaction with the physical asset, but rather reflects its state for monitoring, analysis, and prediction purposes [4]. Typical examples include telemetry systems in motorsport, where data transmitted from a vehicle are analyzed in real time without directly affecting its operation.

1.2 Problem Statement: Data Loss and Disconnections in IoT Environments

For a digital shadow to be reliable, the underlying data flow must be **continuous and consistent**. In practice, however, this condition is difficult to guarantee. Sensor degradation, environmental noise, and network instability can all introduce disruptions in the data stream.

Network reliability represents a major source of issues: interference, congestion, or electromagnetic disturbances may lead to packet loss, which manifests as missing data points in time series. In IoT communication protocols such as MQTT, delivery guarantees depend on the selected quality-of-service level, and message loss may still occur under unstable network conditions [3]. In real-time systems, even delays on the order of tens of milliseconds may affect the responsiveness of time-sensitive applications.

Additionally, power outages or hardware failures may produce extended gaps in the collected data. These interruptions can be interpreted as missing data in time series, a well-known problem that may significantly affect downstream analysis and predictive models [5]. Without appropriate mitigation strategies, these discontinuities compromise the integrity of the digital shadow, limiting its usefulness for both monitoring and forecasting tasks.

1.3 Research Objectives and Requirements

1.3.1 Objectives

The main objective of this thesis is to design and validate an IoT framework capable of ensuring **data continuity** in a digital shadow, even in the presence of temporary disconnections. Specifically, the work aims to:

- Design a modular architecture for real-time acquisition, distribution, and persistence

of measurements, leveraging scalable and lightweight technologies such as MQTT for communication and time-series databases for storage.

- Define a workload model that formally describes the number of messages generated by multiple sensors, and use this model to guide system dimensioning.

- Develop a real-time predictive module capable of generating surrogate values when sensor data are temporarily unavailable, analysing both statistical and neural approaches in terms of accuracy and computational cost.

- Experimentally evaluate the proposed system under realistic conditions, measuring end-to-end latency, throughput, and prediction accuracy.

1.3.2 Functional Requirements

The system must support the ingestion of measurements from a variable number of sensors and their distribution to multiple downstream services. It must detect disconnections and ensure continuity of data availability, even when real measurements are temporarily missing.

The system must support both real-time and aggregated processing modes, enabling prediction at different temporal granularities. It must also allow the integration of heterogeneous forecasting approaches without constraining the choice of specific models.

1.3.3 Non-Functional Requirements

For practical deployment, the framework must be scalable, handling increasing numbers of sensors and higher sampling frequencies without performance degradation. Processing latency must remain below the sampling interval to preserve real-time guarantees.

The solution should be portable and suitable for heterogeneous environments, including edge devices and cloud infrastructures with different computational capabilities. Additionally, robustness and ease of integration of new services are essential design requirements.

1.4 Main Contributions

The main contributions of this work can be summarized as follows:

- **Design of a real-time IoT pipeline**: a modular architecture based on MQTT is proposed, clearly separating publishing and subscribing roles while integrating services for persistence, disconnection monitoring, and prediction. This design enables extensibility without modifying existing components.

- **Workload modelling and scalability analysis**: a simple analytical model, $\lambda = L \times M \times R$, is introduced to estimate the message rate as a function of the number of monitored locations, variables per location, and sampling frequency. This model supports system sizing and performance evaluation.
- **Predictive module with a unified interface**: a forecasting framework is designed and implemented to support heterogeneous models through a common abstraction layer. This enables seamless integration of both statistical (ARIMA) and neural (LSTM) approaches, allowing runtime selection based on accuracy and latency constraints.
- **Comprehensive experimental evaluation**: the system is evaluated in a controlled environment simulating multiple sensors and varying workloads. Metrics such as end-to-end latency, throughput, and message loss are analysed across different pipeline configurations. The predictive module is assessed in terms of accuracy and runtime performance in both streaming and batch scenarios.

1.5 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 presents the *state of the art*, covering IoT monitoring systems, digital shadow and digital twin concepts, communication protocols, and forecasting techniques. Chapter 3 describes the proposed pipeline architecture, detailing its communication, processing, and storage components, as well as the workload model.

Chapter 4 focuses on the design of the predictive module, including data preprocessing, model selection, and the unified interface. Chapter 5 reports the experimental evaluation, analysing system performance under different workloads and comparing forecasting approaches. Finally, Chapter 6 summarizes the findings, discusses limitations, and outlines directions for future work.

Chapter 2

State of the Art

2.1 Executive Summary

IoT deployments frequently experience temporary data disruptions caused by sensor faults, network instability, gateway failures, or energy constraints. When such disruptions occur, downstream components of the data pipeline—such as stream analytics, monitoring dashboards, or machine learning models—may encounter missing values and temporal gaps. These gaps can degrade the reliability of analytics, bias predictive models, or interrupt real-time monitoring processes [6].

Existing research addresses this problem from several complementary perspectives, which can be broadly categorized into three main directions.

The first direction focuses on **pipeline resilience**. These approaches aim to ensure that data are eventually delivered after temporary disconnections by leveraging buffering mechanisms, persistent sessions, and store-and-forward strategies. Technologies such as MQTT v5, Azure IoT Edge, AWS Greengrass, and OPC UA historian access exemplify this approach [7, 8, 9, 10]. While effective in improving reliability, these mechanisms primarily guarantee eventual consistency rather than maintaining continuous real-time data streams.

The second direction addresses the problem of **missing data reconstruction**. A large body of work proposes statistical and machine learning techniques for imputing missing sensor values. Notable examples include deep learning approaches such as BRITS and graph-based models such as GRIN [11, 12]. These methods typically focus on reconstruction accuracy and offline performance, and may not explicitly consider runtime constraints or integration within real-time data pipelines.

The third direction focuses on **streaming analytics infrastructures**. Stream processing systems such as Kafka and Flink provide mechanisms for maintaining consistent state, handling late data, and updating previously processed records through changelog or upsert semantics [13, 14]. These systems improve robustness at the infrastructure level

but do not directly address the problem of missing measurements at the data source.

Despite these advances, an important gap remains insufficiently explored: maintaining **real-time data continuity during disconnections**. Rather than only reconstructing missing data offline, some applications require the data pipeline to continue producing values at the expected sampling cadence.

This motivates the use of **predictive fallback mechanisms**, where forecasting models generate temporary substitute values when real measurements are unavailable. In such scenarios, predictive models must satisfy strict runtime constraints, as inference latency must remain compatible with the sampling interval of the monitored system.

This thesis investigates predictive approaches under explicit latency constraints and proposes a runtime-aware model selection strategy that balances prediction accuracy with computational feasibility.

Before reviewing these research directions in detail, it is useful to clarify the key concepts adopted in this thesis to frame the notion of data continuity under disconnections.

2.2 Definitions and Problem Setting

2.2.1 Data continuity under IoT disconnections

In the context of this thesis, **data continuity** refers to the ability of an IoT data pipeline to provide a usable, time-aligned signal to downstream applications despite temporary interruptions in data acquisition or transmission.

In this work, data continuity is decomposed into three complementary layers.

Delivery continuity (eventual completeness). The system guarantees that measurements are not permanently lost and can be delivered once connectivity is restored. This is typically achieved through mechanisms such as persistent sessions, local buffering, store-and-forward strategies, or historical backfill retrieval [7, 8, 9].

Stream continuity (real-time regularity). The system continues producing a value at each expected sampling instant even when the original measurement is unavailable. In this case, substitute values—such as forecasts or imputed estimates—are generated to maintain a continuous stream, preserving the assumptions of downstream real-time analytics and monitoring systems.

Semantic continuity (quality-aware meaning). The system preserves information about the origin and quality of each emitted value. For example, industrial standards such as OPC UA associate a quality status with each data value (e.g., Good, Uncertain, or Bad),

allowing downstream components to distinguish between measured values, predictions, and reconstructed data [10].

2.2.2 Missingness mechanisms: MCAR, MAR, MNAR

The characteristics of missing data strongly influence the effectiveness of reconstruction techniques. Following the standard classification introduced by Little and Rubin [6]:

- **MCAR (Missing Completely At Random):** the probability that a value is missing is independent of both observed and unobserved variables.
- **MAR (Missing At Random):** missingness depends on observed variables but not on the missing value itself.
- **MNAR (Missing Not At Random):** missingness depends on the missing value or on unobserved factors.

In IoT systems, missing data frequently occur in the form of **burst gaps**, where several consecutive observations are absent due to temporary disconnections or sensor failures. Moreover, missingness in IoT environments is rarely MCAR. For example, battery depletion, network congestion, or sensor saturation may correlate with specific operating conditions. Consequently, evaluations based solely on random missingness may not accurately reflect real-world deployments.

2.3 Research Directions in Data Continuity

The literature on data continuity in IoT systems can be organized into three main research directions, corresponding to the perspectives introduced in the executive summary.

The following sections analyse these directions in detail, highlighting their strengths and limitations with respect to real-time stream continuity.

2.4 Pipeline Resilience in IoT Systems

Research on resilience mechanisms in IoT systems primarily addresses disconnections as a *transport reliability problem*. The main objective is to ensure that measurements are not permanently lost during temporary network outages by buffering data locally and retransmitting them once connectivity is restored.

These approaches improve **delivery continuity**, but do not necessarily preserve **stream continuity**, since downstream applications may still observe temporal gaps while the system is offline.

2.4.1 MQTT durable sessions and queued delivery

MQTT v5 introduces explicit session expiry semantics via the `Session Expiry Interval`. When configured to persist beyond connection closure, both the client and the broker retain session state, enabling reconnection and continued message delivery [7].

MQTT quality-of-service levels (QoS 0/1/2) further define delivery guarantees at the protocol layer. While these mechanisms improve reliability, they cannot generate values when data production itself is interrupted. Therefore, MQTT durability alone does not ensure **stream continuity**, but only improves **delivery continuity** [7].

2.4.2 Edge buffering and store-and-forward mechanisms

Azure IoT Edge supports operation under intermittent or absent connectivity by buffering telemetry locally and enabling store-and-forward communication. This allows systems to preserve data during disconnections and forward it once connectivity is restored [8].

Similarly, AWS IoT Greengrass provides a Stream Manager component that enables local persistence and controlled export of data streams to cloud services [9].

These mechanisms are effective in preventing permanent data loss and ensuring eventual delivery, but they still result in **temporal gaps during offline periods**, since data are not available to downstream components until connectivity is restored.

2.4.3 Industrial approaches: OPC UA historian and data quality

OPC UA defines a rich data model where values are associated with quality indicators through `DataValue` structures and `StatusCode` fields. This allows consumers to distinguish between valid, uncertain, or invalid data [10].

In addition, OPC UA Part 11 specifies historical access services such as `HistoryRead`, enabling retrieval of past data, including timestamps and quality metadata. These mechanisms support backfilling of missing intervals after connectivity is restored [10].

2.4.4 Limitations of infrastructure-based approaches

While these solutions significantly reduce permanent data loss and support eventual completeness, they present several limitations in real-time scenarios.

First, they do not provide substitute values during disconnections, resulting in gaps in the data stream and breaking **stream continuity**.

Second, they lack explicit mechanisms for reconciling real measurements with any temporary estimates that may be generated during outages.

Finally, they do not explicitly consider the relationship between sampling frequency and latency constraints, which is critical in real-time IoT systems.

Taken together, these limitations indicate that infrastructure-based approaches alone are insufficient to guarantee continuous data availability in real time. This motivates the need for complementary mechanisms capable of generating substitute values during outages, such as predictive models integrated within IoT data pipelines.

2.5 Missing Data Reconstruction Techniques

A second line of research focuses on reconstructing missing measurements using statistical or machine learning methods. In contrast to resilience mechanisms, these approaches directly estimate missing values from historical observations, making them conceptually closer to the goal of maintaining data continuity.

However, in real-time IoT data pipelines, imputation methods must satisfy additional constraints beyond prediction accuracy. In particular, they must support online operation, maintain low computational overhead, and remain feasible on constrained edge hardware.

To better understand their practical implications, existing approaches can be grouped into three categories based on their computational characteristics and suitability for real-time IoT environments: (i) classical and batch-oriented methods, (ii) deep learning-based approaches, and (iii) streaming and edge-aware models.

2.5.1 Classical and batch-oriented methods under edge constraints

Well-established techniques such as mean/last observation, kNN imputation, MICE, and missForest are widely used baselines. However, their runtime can vary significantly on constrained devices. In an edge-focused comparison on a Raspberry Pi 4B, execution times ranged from approximately 0.02s (mean imputation) to 16s (MICE), 18s (missForest), and 38s (kNN), illustrating that accuracy-oriented batch methods may be infeasible for tight sampling intervals [15].

These methods provide useful baselines, but their computational cost makes them unsuitable for real-time operation under strict latency constraints. This observation highlights the need to treat **latency as a first-class feasibility constraint** rather than an implementation detail.

2.5.2 Deep learning-based imputation methods

Deep learning approaches aim to improve reconstruction accuracy by learning complex temporal and spatial dependencies.

BRITS models missing values in multivariate time series using a bidirectional recurrent formulation, explicitly aiming to avoid strong parametric assumptions [11]. However, bidirectionality generally implies non-causal reliance on future context. While this can improve reconstruction accuracy in offline or buffered settings, it conflicts with strict

real-time causal constraints, where future observations are not available at inference time, unless additional buffering is introduced (which in turn increases latency).

Similar limitations apply to other deep learning approaches, including graph-based models such as GRIN, which leverage relational structures among sensors to improve reconstruction quality [12]. As with many high-capacity models, the main challenge lies in bounding inference latency and memory footprint within the sampling interval on target edge hardware. Many works emphasize accuracy gains but do not systematically report latency across representative deployment environments.

2.5.3 Streaming and edge-aware imputation methods

More recent approaches attempt to move from offline reconstruction toward real-time applicability.

MPIN formalizes online imputation of sensor data streams as a continuous process operating over time windows, reconstructing missing values while optimizing both accuracy and efficiency [16]. This class of methods represents a step toward streaming-compatible imputation. However, from a data continuity perspective, an additional requirement remains: the imputation mechanism must be explicitly coupled with the pipeline's end-to-end latency budget and integrated with resilience, backfill, and reconciliation mechanisms.

In parallel, work targeting IoT gateways explores lightweight models that can run with small memory footprints and low execution time, enabling "on-the-fly" imputation before forwarding data upstream. For example, neural regression models have been reported to operate within tight memory constraints, supporting feasibility on edge devices [17]. While promising, these approaches are often limited in scope and lack integration with broader system-level continuity strategies.

2.5.4 Critical perspective

Imputation techniques can reconstruct missing values, but several limitations remain.

First, they often treat imputation as a data preprocessing problem rather than a system-level continuity mechanism.

Second, evaluations are frequently based on simulated missingness patterns that may not reflect real IoT outage dynamics, particularly in the presence of MNAR behavior and burst gaps.

Third, latency and resource constraints on representative edge devices are often under-reported.

Finally, these approaches typically omit reconciliation mechanisms when true values become available through backfill.

Taken together, these observations highlight a fundamental limitation: although imputation techniques aim to reconstruct missing values, they are typically not designed

as system-level solutions for maintaining real-time data continuity. In particular, they rarely account for strict latency constraints, integration with streaming infrastructures, and the need for reconciliation when true measurements become available.

2.6 Streaming Data Processing and Reconciliation

Streaming analytics frameworks provide the infrastructure on which real-time IoT data pipelines operate. These systems define how data streams are processed, how state is maintained, and how previously computed results can be updated when delayed or corrected data become available.

Examples include distributed messaging systems such as Kafka and stream processing engines such as Apache Flink. These platforms support mechanisms such as keyed retention, changelog streams, and upsert semantics, which enable consistent state updates in the presence of late or out-of-order data.

To clarify their role in data continuity, these systems can be analysed along two main dimensions: (i) state consistency and reconciliation mechanisms, and (ii) execution models and runtime constraints.

2.6.1 State consistency and reconciliation mechanisms

Kafka log compaction provides a retention mechanism that guarantees storing at least the *last known value* for each message key within a topic partition, enabling state reconstruction and changelog-style processing [18].

From a data continuity perspective, this could enable a reconciliation strategy. Records can be keyed by `(sensor_id, timestamp)`: during outages, predicted values may be emitted, while later backfilled measurements can be published with the same key, allowing downstream consumers to converge to the latest (true) value.

Similarly, Apache Flink formalizes streaming-to-table equivalence through the concept of *dynamic tables*. With a unique key, a table can be represented as a changelog stream containing INSERT, UPDATE, and DELETE operations [13].

In this model, reconciliation can be expressed as an update operation: predicted values can be inserted with appropriate provenance metadata, and subsequently updated when real measurements arrive, ensuring a consistent and unified state view for downstream applications.

2.6.2 Execution models and latency trade-offs

A second important aspect concerns how streaming systems execute computations under latency and throughput constraints.

Modern stream processing engines often adopt micro-batch or hybrid execution models, balancing throughput, fault tolerance, and latency [14]. These architectural choices directly impact the feasibility of real-time data continuity mechanisms.

In particular, the effectiveness of predictive fallback strategies depends not only on model inference latency, but also on the overhead introduced by the streaming engine itself, including state management, scheduling, and data propagation delays. As a result, system-level latency must be considered in addition to model-level performance.

2.6.3 Runtime scheduling and adaptation

Streaming systems increasingly incorporate runtime adaptation mechanisms, including dynamic scaling, load balancing, and resource-aware scheduling [14].

These capabilities are particularly relevant in IoT environments, where disconnections may lead to bursts of delayed or backfilled data, causing sudden changes in workload. Efficient scheduling and resource management are therefore essential to maintain system stability and ensure timely processing of both real-time and delayed data streams.

2.6.4 Critical perspective

Streaming infrastructures provide strong guarantees for state consistency, late data handling, and result updates. However, they do not, by themselves, define a complete solution for real-time data continuity.

In particular, they do not specify when and how predictive substitute values should be generated during data outages, how to enforce feasibility constraints with respect to sampling intervals, or how to evaluate and select predictive models under runtime and hardware constraints.

Furthermore, while reconciliation mechanisms are supported at the infrastructure level, they must be explicitly integrated with data provenance and quality semantics to distinguish between measured, predicted, and backfilled values.

These limitations highlight that, although streaming systems provide essential building blocks, they must be combined with predictive and runtime-aware components to achieve full data continuity in real-time IoT pipelines.

2.7 Research Gap: Real-Time Predictive Fallback Under Latency Constraints

The literature reviewed in the previous sections highlights a clear separation between three research directions: resilience mechanisms that ensure eventual delivery, imputation

methods that reconstruct missing values, and streaming systems that manage continuous data processing and state updates.

However, relatively little work explicitly addresses the problem of maintaining **real-time stream continuity** during temporary IoT disconnections.

In many applications, downstream components expect data to arrive at a fixed sampling cadence. Interruptions in the measurement stream can therefore degrade monitoring dashboards, anomaly detection pipelines, or other online analytics components. In such scenarios, continuity requires the system to continue producing values even when real measurements are temporarily unavailable.

While existing approaches either ensure eventual delivery or focus on offline reconstruction accuracy, they do not provide a unified solution that guarantees continuity under real-time constraints.

This thesis addresses this gap by investigating **forecasting-based predictive fallback** as a mechanism for preserving data continuity during disconnections. In this setting, predictive models are used to generate temporary substitute values that maintain the expected sampling cadence of the data stream.

This introduces a key requirement that is often underemphasized in the literature: predictive models must not only be accurate, but also computationally feasible within the temporal constraints of the system.

2.7.1 Runtime Feasibility Constraint

In real-time IoT environments, predictive models must operate under strict latency constraints. Let T_s denote the sampling interval of the sensor stream and $\mathcal{L}(m)$ the inference latency of model m on the target hardware.

For a model to be usable in a predictive fallback mechanism, its inference latency must satisfy the following feasibility condition:

$$\mathcal{L}(m) \leq T_s \tag{2.1}$$

This condition expresses a fundamental principle: a predictive model is only deployable if it can generate its output within the time available between two consecutive observations. Consequently, model evaluation must consider both prediction accuracy and runtime performance on the target hardware.

2.7.2 Reconciliation and Data Provenance

A continuity-oriented mechanism must also address the reconciliation phase that follows the restoration of connectivity.

During disconnections, predicted values may be emitted to preserve stream continuity.

When real measurements become available through backfill, the system must update or replace the provisional values previously generated.

Streaming infrastructures provide mechanisms that can support this process, but their integration with predictive components must be explicitly designed. For example, Kafka log compaction enables state convergence by retaining the latest value associated with each key, while systems such as Apache Flink support update semantics through changelog streams and upsert operations [18, 13].

In addition, maintaining data provenance is essential. Each emitted value should explicitly indicate whether it corresponds to a measured observation, a predicted substitute, or a backfilled update. Industrial standards such as OPC UA adopt this principle by associating quality metadata with each transmitted value [10].

Taken together, these requirements highlight the need for an integrated approach that combines predictive modeling, runtime-aware decision-making, and streaming reconciliation mechanisms to fully ensure real-time data continuity to ensure real-time data continuity in IoT systems.

In this work, the focus is placed on predictive fallback under real-time constraints. While reconciliation mechanisms are discussed as a fundamental requirement for complete data continuity, they are not explicitly implemented. However, the proposed architecture is designed to support their integration as future extensions.

2.8 Evaluation Methodology and Metrics

To assess data continuity under disconnections, both system-level and model-level metrics must be considered. This distinction reflects the dual nature of the problem: ensuring that the data pipeline remains operational under varying conditions, while also guaranteeing that predictive components are both accurate and computationally feasible.

2.8.1 System-level metrics

System-level metrics capture the behavior of the overall data pipeline under normal operation, disconnections, and recovery phases.

Throughput. Number of records processed per second by ingestion and stream processing components. This metric indicates whether the system can sustain both steady-state operation and recovery bursts after reconnection.

End-to-end latency. Time between data generation and availability for downstream applications. In continuity-oriented pipelines, latency may differ for measured values, predicted substitutes, and backfilled updates, making it necessary to analyse these components separately.

Gap statistics. Frequency, duration, and distribution of missing data segments. Burst

gaps are particularly relevant in IoT environments and strongly influence the effectiveness of reconstruction or fallback strategies.

2.8.2 Model-level metrics

Model-level metrics evaluate the quality and feasibility of predictive components operating within the pipeline.

Prediction accuracy. Metrics such as MAE or RMSE computed specifically on missing segments or simulated outages, reflecting the ability of models to approximate true measurements.

Runtime feasibility. Inference latency, memory footprint, and CPU utilization on the target hardware. These metrics determine whether a predictive model satisfies the feasibility condition introduced in Section 2.7.

Impact on downstream analytics. Evaluation of whether predicted values preserve the stability and reliability of downstream components, such as anomaly detection systems or monitoring dashboards.

Taken together, these metrics enable a comprehensive evaluation of data continuity mechanisms, capturing both system-level robustness and model-level performance under real-time constraints.

2.9 Comparative Summary of Representative Works

Table 2.1 summarizes the main capabilities and limitations of representative approaches discussed in this chapter. The comparison confirms that existing solutions typically address only part of the continuity problem, focusing either on delivery reliability, missing data reconstruction, or state reconciliation, but rarely integrating all these aspects under explicit real-time constraints.

Reference	Type	Main Idea	Online	Missing Data	Reconciliation
		Messaging and Edge Platforms			
MQTT v5 Spec [7]	Messaging protocol	QoS levels and persistent sessions for reliable telemetry delivery	Yes	No	No
Azure IoT Edge [8]	Edge platform	Store-and-forward buffering and offline execution of edge workloads	Yes	No	Partial
AWS Greengrass [9]	Edge platform	Local stream storage and export to cloud services	Yes	No	Partial
OPC UA Core [10]	Industrial data standard	Data quality flags and historian access for historical measurements	Yes	No	Backfill
		Imputation Models			
Erhan et al. [15]	Edge imputation study	Evaluation of classical imputation methods on IoT devices	Limited	Yes	No
França et al. [17]	Gateway ML model	Lightweight neural models for on-device missing data reconstruction	Yes	Yes	No
BRITS [11]	Deep learning model	Bidirectional RNN architecture for time-series imputation	Limited	Yes	No
GRIN [12]	Graph neural network	Spatio-temporal imputation using sensor relationships	Limited	Yes	No
MPIN [16]	Stream imputation model	Window-based online imputation for sensor streams	Yes	Yes	No
		Streaming and State Reconciliation			
Kafka log compaction [18]	Streaming infrastructure	Keyed retention enabling state convergence	Yes	No	Yes
Flink dynamic tables [13]	Stream processing engine	Changelog streams and upsert semantics for state updates	Yes	No	Yes

Table 2.1: Comparison of representative approaches for IoT data continuity, highlighting their capabilities in terms of online operation, missing data handling, and support for reconciliation mechanisms.

2.10 Synthesis

The literature reviewed in this chapter highlights three complementary research directions.

The first direction focuses on improving the reliability of IoT data pipelines through buffering, persistent sessions, and backfill mechanisms. These approaches reduce the risk of permanent data loss and enable eventual delivery of measurements.

The second direction addresses the reconstruction of missing data through statistical and machine learning techniques. While these methods can estimate missing values with reasonable accuracy, their applicability in real-time IoT environments is often constrained by computational cost, causality requirements, and hardware limitations.

The third direction studies streaming analytics infrastructures and their ability to manage state updates, late data, and consistent processing through mechanisms such as changelog streams and keyed retention.

Taken together, these works provide valuable building blocks but rarely address the problem of maintaining real-time data continuity during temporary disconnections. This observation motivates the predictive and runtime-aware approach developed in the remainder of this thesis.

Chapter 3

IoT Framework Design

The previous chapters introduced the main challenges associated with the management of IoT data streams and motivated the use of predictive models to compensate for missing measurements during temporary sensor disconnections.

To support this capability, a data infrastructure is required to collect sensor measurements, manage real-time data streams, and provide reliable access to historical observations. This chapter therefore presents the architecture of the IoT framework developed in this work.

The proposed framework implements a **modular data pipeline** responsible for acquiring measurements from distributed sensors, transporting them through a publish/subscribe communication infrastructure, and storing them in a persistent time-series database where they can be accessed by monitoring and analytical services.

Although the architecture is designed to remain independent from a specific application domain, the experimental validation presented in this work considers an environmental monitoring scenario involving a smart-room equipped with sensors measuring variables such as temperature and humidity. While this setup provides a realistic source of data streams, the architectural design itself remains applicable to other sensor-based environments with minimal modifications.

The remainder of this chapter is organized as follows. Section 3.1 introduces the overall system architecture and the main components involved in the data pipeline. Section 3.2 presents the workload model used to reason about system scalability. The following sections describe the communication layer, the subscriber processing model, the persistence layer, and the visualization infrastructure. Finally, the mechanism used to detect sensor disconnections and activate the predictive fallback mode is presented.

3.1 Architecture Overview

The proposed framework is organized as a **modular data pipeline** in which sensor measurements are progressively processed by different components of the system.

At the beginning of the pipeline, sensing devices periodically generate environmental measurements and publish them to a message broker. The broker acts as the central communication hub of the system and distributes incoming messages to all subscribed services.

Downstream components operate as independent consumers of the data stream. Each service subscribes to the topics relevant to its specific task and processes the received measurements without interfering with other components of the infrastructure.

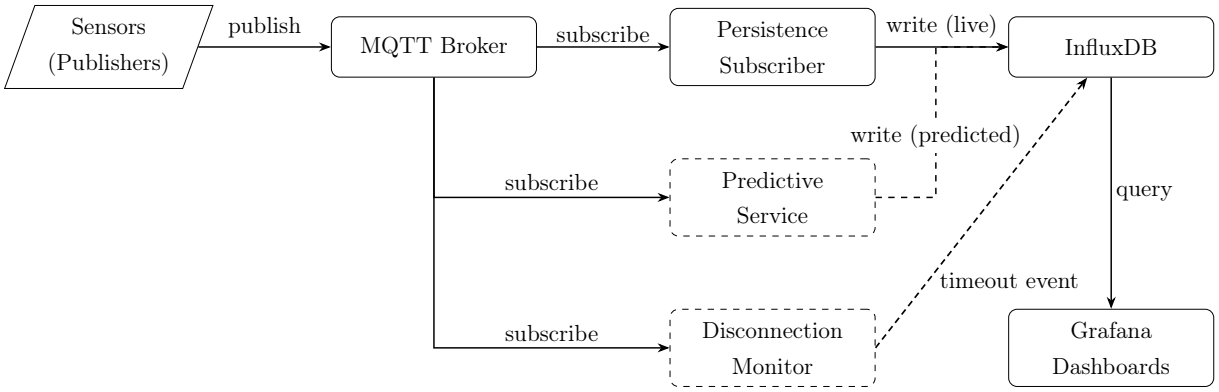

Figure 3.1: Overview of the proposed IoT architecture. The diagram highlights the separation between the main layers of the system, including data producers (sensors), the MQTT communication broker, downstream subscriber services, the persistence layer, and the visualization interface.

Figure 3.1 illustrates the overall architecture of the framework. Sensors act as data producers that periodically publish measurements to the MQTT broker. Multiple subscribers consume the same data stream and perform different tasks such as data persistence, monitoring of the data flow, or predictive analysis.

The persistence service is responsible for storing incoming measurements in the time-series database, while additional services can process the same stream to detect anomalies, monitor system behaviour, or generate predictive estimates when live measurements become temporarily unavailable.

This modular organization enables new services to be integrated without modifying the existing data pipeline. As a result, the framework remains extensible and can support additional analytical modules or monitoring components with minimal architectural changes.

3.2 System Workload Model

In order to reason about the scalability of the proposed architecture, it is useful to introduce a simplified model describing the volume of data generated by the sensing infrastructure.

Let

- L denote the number of monitored locations (or sensing units),
- M the number of variables collected per location,
- R the sampling rate of each variable (expressed in messages per second).

Under these assumptions, the global message rate produced by the system can be approximated as

$$\lambda = L \times M \times R$$

where λ represents the total number of messages per second generated by the sensing infrastructure.

This formulation assumes that all variables are sampled at approximately the same rate R. While real deployments may involve heterogeneous sampling frequencies, this simplified model provides a convenient approximation for reasoning about the overall system workload. In cases where different variables are sampled at different rates, λ can be interpreted as the aggregate message rate generated by the system.

Under this interpretation, λ provides a compact way to reason about the workload experienced by the different components of the data pipeline. In particular, it determines the amount of traffic handled by the communication infrastructure, the processing load imposed on subscriber services, and the write rate sustained by the persistence layer.

Although the prototype implementation considered in this work operates under moderate workloads, the proposed model provides a useful reference for discussing how the architecture behaves as the number of sensors or the sampling frequency increases.

The following sections use this workload formulation to analyze the design choices adopted in the communication, processing, and persistence layers of the framework.

3.3 Communication Layer

The communication layer is responsible for transporting measurements generated by distributed sensing devices to the components responsible for storage, monitoring, and predictive analysis.

A key requirement of the proposed framework is the ability to decouple data producers from data consumers. Sensors should be able to publish measurements without knowledge of which services will process them, while multiple downstream components should be able to consume the same data stream independently.

To satisfy these requirements, the framework adopts a **publish/subscribe communication model**. In this paradigm, sensors act as publishers that transmit measurements to a message broker, while downstream services subscribe to the topics of interest and process incoming messages according to their specific role within the system.

Among the available publish/subscribe protocols, MQTT was selected due to its lightweight message overhead, broker-based routing mechanism, and widespread adoption in IoT deployments. The broker acts as the central communication hub of the system, receiving measurements from publishers and distributing them to all subscribed consumers.

From a system perspective, the broker represents a critical component of the communication infrastructure, as it mediates the distribution of the entire message stream generated by the sensing system. The performance of the broker therefore influences the maximum throughput that the communication layer can sustain.

However, the internal implementation and scalability mechanisms of the broker fall outside the scope of this work. In the proposed framework, the broker is treated as an external infrastructure component, while the focus of the analysis is placed on the behaviour of subscriber services and their ability to process incoming data streams.

Even though the broker handles the routing, the way data streams are organized into topics plays a key role in how messages are distributed across subscribers. Therefore, the design of topic structure becomes a central element in enabling flexible data routing and control over distribution.

3.3.1 Topic Organization

Once a publish/subscribe communication model is adopted, an important design decision concerns how measurements should be organized within MQTT topics.

The topic structure determines how easily subscribers can access specific subsets of the data stream and therefore influences the distribution of processing load across consumer services.

The simplest approach consists of publishing all measurements under a single topic. In this configuration, all subscribers receive the entire message stream and must locally inspect each message in order to determine whether it is relevant to their task.

While this design minimizes topic management complexity, it introduces a potential scalability limitation. As the global message rate λ increases, subscribers may receive a large number of messages that are not relevant to their processing task, leading to unnecessary processing overhead.

To address this limitation, measurements can be partitioned across multiple topics according to different routing dimensions.

A common strategy consists of organizing topics based on the monitored variable (e.g., `/project/temperature`) or based on the monitored location (e.g., `/project/room1`). This organization allows subscribers to selectively access only the portion of the data stream relevant to their processing logic.

A more flexible solution consists of adopting a **hierarchical topic** structure that exposes both routing dimensions explicitly:

`/project/location/variable`

This hierarchical organization makes it possible to design subscriber services with increasingly fine-grained responsibilities. By subscribing to a specific location–variable pair (e.g., `/project/room1/temperature`), a consumer can operate exclusively on the data stream generated by a single sensor. In this configuration, the subscriber receives only the measurements that are strictly relevant to its processing task, reducing the amount of incoming traffic and allowing the implementation of highly specialized processing pipelines.

Such a design is particularly beneficial for analytical components that operate on individual time series. For example, forecasting models trained on historical measurements of a specific sensor may subscribe directly to the corresponding location–variable stream, ensuring that the prediction service processes only the data required for that model.

However, this level of specialization also introduces architectural trade-offs. If a dedicated subscriber is created for every location–variable pair, the number of consumer services grows proportionally to the number of sensing streams in the system.

Nevertheless, the hierarchical topic structure does not require the system to operate at such a level of specialization. Instead, it provides the flexibility to adopt different subscription granularities depending on the needs of each service. Through MQTT wildcard subscriptions (e.g., `+/temperature` or `room1/#`), subscribers may still operate on aggregated subsets of the data stream, preserving loose coupling while maintaining flexible routing.

As a result, hierarchical topic organization does not replace previous approaches but rather generalizes them. It enables the creation of highly specialized subscribers when needed, while still allowing broader subscriptions that preserve loose coupling and simplify system management.

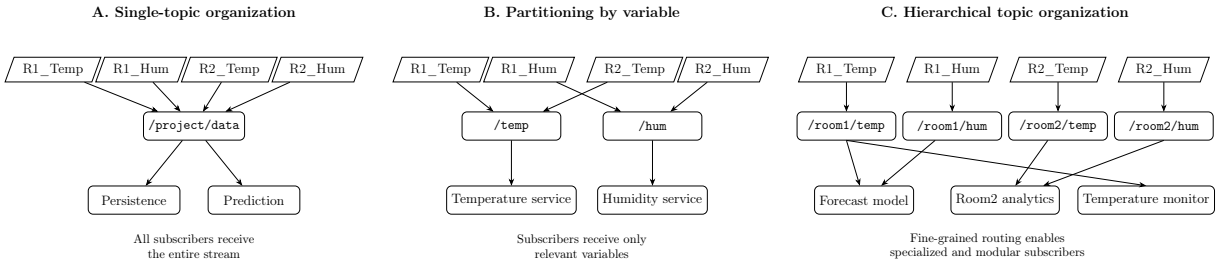

Figure 3.2: Comparison of alternative MQTT topic organizations. In a single-topic design, all subscribers receive the entire message stream and must locally filter irrelevant data. Partitioning by variable reduces unnecessary message processing by allowing services to subscribe only to relevant variables. A hierarchical topic structure further increases flexibility by enabling subscriptions at different granularity levels, from single location–variable streams to aggregated views by location or variable.

Figure 3.2 summarizes the main differences between alternative topic organizations and illustrates how the chosen structure influences the distribution of messages across subscriber services.

In summary, the organization of MQTT topics contributes to improving the distribution of the message load by allowing subscribers to access only the portions of the data stream relevant to their task. However, this is not its primary purpose, and the system may still experience performance limitations when message arrival rates exceed the processing capacity of subscriber services.

For this reason, the efficiency of the subscriber-side processing pipeline remains a critical factor for system scalability. The next subsection therefore examines the potential bottlenecks that may arise during message processing.

3.3.2 Subscriber Processing Bottleneck

Even when topic organization distributes different data streams across multiple subscribers, the processing capacity of each consumer remains a critical factor for the scalability of the system.

In typical MQTT client implementations, incoming messages are delivered to the subscriber through a callback function that is triggered whenever a new message arrives. This callback is responsible for executing the processing logic associated with the received message, which may include operations such as message parsing, validation, or persistence to a database.

If message processing is performed sequentially within the callback, the throughput of the subscriber becomes limited by the time required to process each message.

When the arrival rate of messages exceeds the processing capacity of the subscriber, incoming messages begin to accumulate in the internal queue of the client. As a consequence, messages experience increasing waiting times before being processed, leading to growing end-to-end latency.

In this scenario, the bottleneck is no longer determined by the communication infrastructure or by the organization of MQTT topics, but rather by the ability of the subscriber to process incoming messages at a sufficient rate.

For this reason, improving the efficiency of the subscriber-side processing pipeline becomes essential in order to sustain higher message rates.

3.3.3 Concurrent Message Processing

To mitigate the processing bottleneck described in the previous section, the proposed architecture adopts a concurrent message processing strategy that decouples message reception from message processing.

Instead of performing all operations directly within the MQTT callback, the callback is limited to lightweight tasks such as extracting message metadata and placing the received message into an internal processing queue.

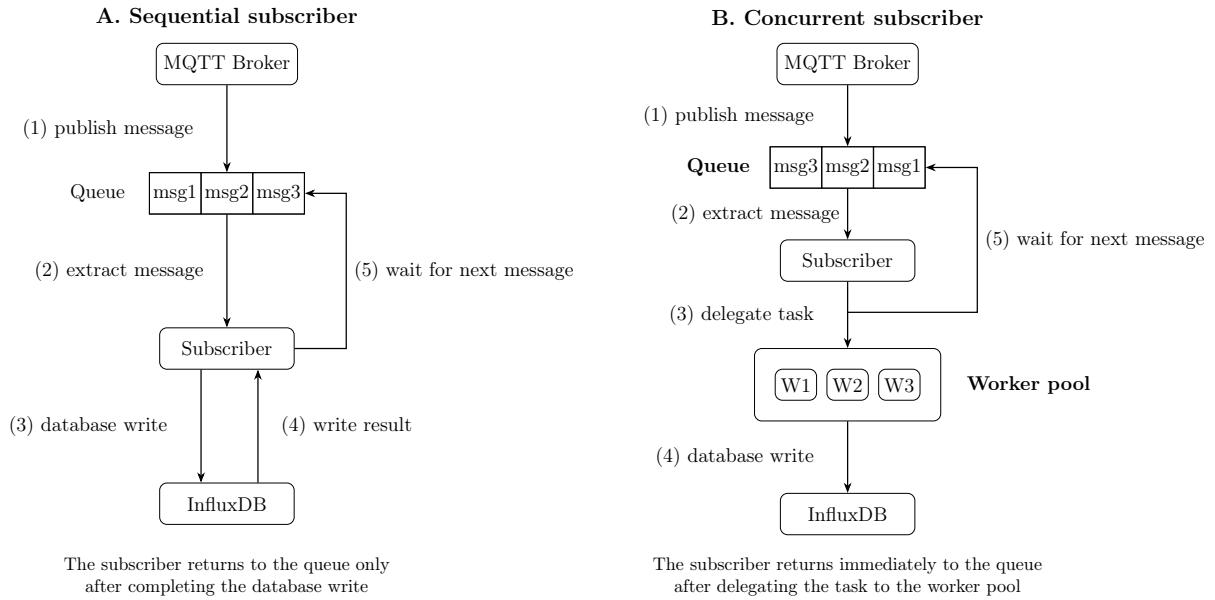

Figure 3.3: Comparison between sequential and concurrent subscriber-side message processing. In the sequential design, the subscriber extracts a message from the queue, processes it, writes it to the database, receives the write result, and only then returns to the queue. In the concurrent design, the subscriber dequeues the message and delegates the task to a worker pool, allowing it to immediately resume message reception while workers handle database writes asynchronously.

The actual processing of the message is then delegated to a pool of worker threads responsible for executing the required operations, such as data validation and database persistence.

This design allows multiple messages to be processed concurrently, significantly increasing the effective throughput of the subscriber. While the MQTT client continues receiving incoming messages, worker threads independently process queued tasks without blocking the reception pipeline.

As a result, the system becomes capable of sustaining higher message rates while reducing the risk of queue buildup and increasing end-to-end latency.

However, the size of the worker pool must be carefully selected. An insufficient number of workers may fail to alleviate the processing bottleneck, while an excessively large thread pool may introduce scheduling overhead and resource contention. The optimal configuration therefore depends on the computational resources available and on the complexity of the processing tasks executed by each worker.

3.4 Persistence Layer

The persistence layer is responsible for storing incoming sensor measurements in a structured and queryable form, enabling both real-time visualization and historical analysis.

Given the continuous stream of measurements generated by the sensing infrastructure,

the storage system must sustain a continuous ingestion workload while supporting efficient time-based queries. For this purpose, the framework adopts InfluxDB, a NoSQL database specifically designed for time-series data.

InfluxDB is optimized for high write throughput and for queries that operate on temporal windows, making it particularly suitable for applications involving continuous monitoring and real-time analytics.

3.4.1 Time-Series Representation in InfluxDB

In InfluxDB, measurements are stored as time-series points organized within a bucket. Each point represents a single observation and is defined by three main components:

- **Measurement name**, identifying the logical time series to which the point belongs;
- **Fields**, representing the actual measured values;
- **Tags**, providing indexed metadata used for filtering and grouping data.

In the proposed framework, each incoming MQTT message is translated into a single point written to the database. The measurement name identifies the monitored variable (e.g., `temperature`), while contextual information such as the location of the sensor is represented as a tag.

For example, a temperature measurement collected in `room1` is stored as a point belonging to the `temperature` measurement series, with the corresponding room identifier stored as a tag. The numeric value of the measurement is stored as a field.

This representation allows the database to maintain independent time series for each variable while still enabling efficient queries across locations through tag-based filtering.

3.4.2 Write Workload

The ingestion workload experienced by the database is directly determined by the global message rate introduced earlier in the chapter:

$$\lambda = L \times M \times R$$

where L denotes the number of monitored locations, M the number of variables per location, and R the sampling frequency.

Since each received message produces a single database record, the expected write rate is approximately equal to λ writes per second.

If t_w denotes the average latency of a database write operation, the persistence layer can sustain the incoming workload as long as

$$\lambda \cdot t_w < 1$$

Otherwise, write operations may accumulate in the processing queue, increasing the overall latency of the ingestion pipeline.

3.4.3 Write Strategy and Scalability Considerations

As discussed in Section 3.3.2, database writes may represent a significant portion of the processing time associated with each incoming message. As the global message rate increases, the persistence stage may therefore become one of the dominant contributors to the overall latency of the ingestion pipeline.

Several techniques can be employed to mitigate this limitation, including batching multiple points into a single write request or introducing asynchronous buffering mechanisms that decouple message processing from database persistence.

However, the behaviour of these mechanisms may depend on internal buffering policies and optimization strategies implemented by the database engine. As a consequence, the effective write throughput under high ingestion workloads may be influenced by factors that are not directly controlled by the application layer.

For this reason, the current prototype adopts a conservative synchronous write strategy. Each message is written to the database individually, and the subscriber waits for the confirmation returned by the database before continuing the processing pipeline.

Although InfluxDB provides native support for asynchronous and batched writes, the synchronous approach keeps the ingestion pipeline fully observable and simplifies the interpretation of system behaviour. In particular, it ensures that the measured processing latency primarily reflects the cost of message processing and database persistence rather than additional optimizations introduced by internal buffering mechanisms.

More advanced write strategies may be considered in future deployments operating under higher ingestion workloads, where throughput becomes a more critical constraint.

3.5 Visualization Layer

The visualization layer provides real-time monitoring and exploratory analysis capabilities on top of the persisted time-series data. It represents the primary interface for inspecting sensor measurements, identifying anomalies, and validating the behaviour of the predictive module.

In the proposed framework, visualization is implemented using Grafana, a widely adopted platform for time-series dashboards. Grafana integrates directly with InfluxDB

and retrieves measurements through time-based queries, enabling interactive exploration of the collected data over different temporal windows.

3.5.1 Query Model and Aggregation

Sensor data are retrieved from InfluxDB through time-range queries. For short time intervals, raw measurements can be displayed directly without significant performance impact.

However, when larger time windows are selected, the number of returned points grows proportionally to the duration of the query interval:

$$N = \lambda \times T$$

where T is the selected time range and λ is the global message rate introduced earlier in the chapter.

To maintain interactive dashboard responsiveness, window-based aggregation is applied. Measurements are averaged over fixed temporal windows, reducing the number of displayed points while preserving the overall evolution of the signal. The aggregation window size is dynamically adjusted depending on the selected time range, enabling a trade-off between temporal resolution and query performance.

3.5.2 Integration with Predictive Outputs

The adopted data model allows predicted values to be stored using the same schema as live measurements, with an additional tag indicating their origin (e.g., `source=predicted`).

This design enables dashboards to seamlessly display:

- live measurements,
- predicted values,
- direct comparisons between observed and forecasted signals.

Since predicted data follow the same storage structure as live measurements, the visualization layer does not require structural modifications when the predictive module is activated. This further demonstrates the modularity of the proposed architecture.

3.5.3 Limitations

The current visualization layer is primarily intended for monitoring and experimental validation. As the system scales and the global message rate λ increases, visualization performance may increasingly depend on database-side aggregation policies and retention strategies.

3.6 Disconnection Detection and Predictive Mode

A key objective of the proposed framework is to preserve data availability at the application level even when live sensor measurements become temporarily unavailable. This is achieved through an automatic transition to a predictive operating mode.

3.6.1 Disconnection Detection Model

Let R denote the expected sampling rate of a given metric. The expected inter-arrival time between consecutive measurements is therefore

$$\Delta t_{expected} = \frac{1}{R}$$

A disconnection event is detected when the time elapsed since the last received measurement exceeds a predefined threshold:

$$t_{now} - t_{last} > k \cdot \Delta t_{expected}$$

where k is a configurable tolerance factor that accounts for network jitter and small transmission delays.

This detection mechanism assumes that the nominal sampling rate is known a priori, which is consistent with the experimental setup considered in this work.

3.6.2 System States and Mode Switching

The system operates as a two-state machine:

- **Live Mode** – incoming sensor measurements are received through MQTT and persisted to the database.
- **Predictive Mode** – forecasted values are generated in place of missing measurements.

State transitions occur as follows:

- Live → Predictive: triggered when a timeout condition is detected.
- Predictive → Live: triggered when new real measurements are received.

3.6.3 Architectural Integration

To preserve modularity, the ingestion pipeline remains independent from the forecasting logic. Live measurements are handled through MQTT and persisted by the ingestion subscriber described in the previous sections.

Disconnection detection is performed by a dedicated monitoring component that subscribes to the same MQTT streams. For each data stream, identified by the pair `(metric, location)`, the monitor tracks the timestamp of the most recent message and triggers a missing-data condition when the timeout threshold is exceeded.

Predicted values are generated by the predictive subsystem and persisted directly to InfluxDB using the same schema as live measurements, with an additional tag indicating their origin (e.g., `source=predicted`). This approach preserves data traceability while allowing dashboards to display continuous time series without modifying the ingestion pipeline.

3.6.4 Design Considerations and Limitations

The proposed detection mechanism assumes a stable and known sampling rate. In more heterogeneous sensing environments, more sophisticated approaches (such as adaptive thresholds or heartbeat signalling) may be required.

Furthermore, the predictive mode currently replaces missing measurements without retroactive correction once live data resume. Future extensions may include reconciliation strategies to align predicted and real measurements.

3.7 Summary

This chapter presented the architecture of the proposed IoT framework supporting data acquisition, storage, and monitoring of sensor measurements.

The system follows a modular publish/subscribe design based on MQTT, where incoming measurements are processed by subscriber services, persisted in a time-series database, and exposed through a visualization layer for real-time monitoring.

The chapter also introduced the mechanisms used to detect temporary sensor disconnections and to activate a **predictive fallback mode** in order to maintain continuity in the observed time series.

Within this architecture, the predictive component operates as an additional processing module that generates surrogate measurements when live data become unavailable.

The internal design of this predictive module and the methodology used to evaluate forecasting models are presented in the next chapter.

Chapter 4

Predictive Module

While Chapter 3 described the system-level architecture of the IoT framework, this chapter focuses on the internal design of the predictive module and the methodology used to evaluate candidate forecasting models under real-time operational constraints.

4.1 Problem Formulation in the IoT Context

The predictive module operates as a continuously running component within the IoT data pipeline introduced in Chapter 3. Its role is to process incoming measurements and generate forecasts in real-time as new observations become available. Its purpose is twofold: (i) **provide short-term surrogate values when live measurements are temporarily unavailable**, and (ii) **enable real-time forecasting capabilities for monitoring and predictive analysis**.

Let x_t denote the value of a metric sampled at regular time intervals Δt. At any time instant t, the forecasting task consists of estimating future values

$$\hat{x}_{t+1}, \ldots, \hat{x}_{t+H}$$

based on the historical observations available up to time t.

4.1.1 Operational Constraints

In an IoT deployment scenario, forecasting accuracy alone is insufficient. The predictive model must also satisfy several operational constraints imposed by the real-time nature of the system:

- **Sampling constraint:** predictions should be produced at the same rate as incoming samples (Δt), or at a lower rate if a batch strategy is adopted.

- **Latency constraint:** the inference time T_{inf} required to generate a forecast must satisfy

$$T_{\text{inf}} \leq L_{\max},$$

  where $L_{\max}$ is the maximum acceptable latency, generally related to the sampling interval.

- **Resource constraint:** memory usage and computational overhead must remain compatible with the intended deployment environment.

These constraints motivate the adoption of a **runtime-aware evaluation methodology**, in which model suitability is determined by both predictive performance and operational feasibility.

4.1.2 Streaming vs Batch Forecasting Modes

Two runtime operating modes can be considered:

- **Streaming (one-step) mode:** the model produces $\hat{x}_{t+1}$ at every sampling instant. This corresponds to the strictest real-time requirement.

- **Batch (multi-step) mode:** the model produces a horizon of H predictions at a lower frequency (e.g., every B samples), trading off update granularity for reduced invocation overhead.

In addition, multi-step forecasting is used as a robustness analysis tool to quantify error accumulation over increasing horizons.

4.2 Data Layer

Time series collected from IoT systems often present several data quality issues that can negatively affect forecasting models. Common problems include inconsistent timestamps, irregular sampling intervals, sensor faults producing out-of-range measurements, and missing segments caused by temporary communication failures or device interruptions.

Before training forecasting models, it is therefore necessary to standardize the dataset to ensure both data quality and structural consistency. This requirement is particularly important in a benchmarking scenario, where multiple models must be evaluated under comparable conditions using the same input data.

To address these challenges, a set of data preparation utilities was developed to support dataset loading and preprocessing. These utilities provide a standardized pipeline that transforms raw CSV time series into a format suitable for forecasting models while maintaining flexibility in handling different dataset characteristics.

The preprocessing stage is implemented as a modular pipeline that standardizes the input time series before model-specific transformations.

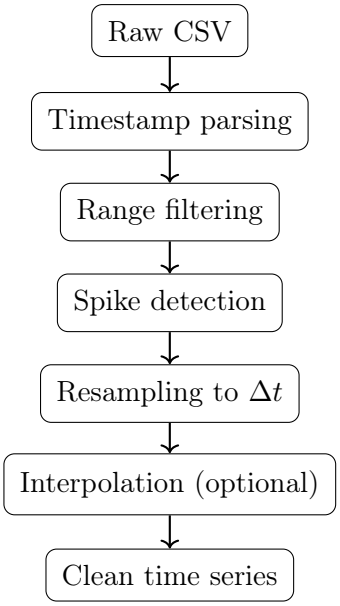

Figure 4.1: Preprocessing pipeline used to standardize raw IoT time series data before forecasting.

The pipeline performs the following steps:

- loading of the raw dataset
- timestamp parsing and chronological ordering
- removal of values outside valid physical ranges
- spike detection based on first differences
- resampling to a fixed temporal resolution
- optional interpolation of missing observations

This process ensures that all forecasting models operate on a consistent and regularly sampled time series representation.

4.2.1 Signal Cleaning

Sensor measurements may occasionally contain unrealistic values due to hardware faults, calibration errors, or temporary disturbances. Such values can significantly degrade the performance of forecasting models if used directly during training.

To mitigate this issue, the preprocessing pipeline allows filtering values that fall outside a predefined valid range. Measurements detected as invalid are replaced with missing values, preventing them from contaminating the learning process.

In addition, optional spike detection can be applied by analyzing the first-order difference between consecutive samples. Discontinuities exceeding a predefined threshold may reflect measurement inaccuracies as sensor noise rather than genuine signal variance. When detected, these anomalies can also be treated as missing values and handled during the subsequent preprocessing stages.

4.2.2 Resampling

IoT datasets frequently exhibit irregular sampling intervals due to network delays, asynchronous sensor transmissions, or temporary outages. However, most forecasting models assume that observations are collected at regular time intervals.

For this reason, the dataset is resampled to a fixed temporal resolution Δt, ensuring that the resulting time series follows a consistent sampling structure. This step guarantees that the dataset representation matches the **operational conditions** under which the predictive module will run at deployment time.

Resampling also simplifies the comparison of different forecasting models, since all models receive input sequences with the same temporal structure.

4.2.3 Missing Data Handling

Missing values are a common occurrence in sensor datasets and may arise from temporary sensor failures, network issues, or data acquisition interruptions. These missing segments must be carefully managed, as they can disrupt the temporal continuity required by forecasting models.

Small gaps can often be reconstructed using interpolation techniques, allowing the time series to maintain a continuous structure without introducing significant distortions. The preprocessing utilities therefore provide an optional interpolation step that can fill short missing intervals.

However, larger missing segments may require more careful consideration. In some cases, interpolating long gaps may introduce artificial patterns that negatively affect the model training process.

To address this issue, the framework allows specifying a maximum number of consecutive missing samples that can be interpolated. When this threshold is exceeded, the missing values are preserved in the dataset, allowing the practitioner to decide how the gap should be handled according to the specific characteristics of the dataset.

Although the current implementation does not automatically resolve large missing segments, the framework exposes configuration parameters that make it possible to control

interpolation limits and inspect the resulting dataset. Future extensions could include automated strategies for handling extended missing intervals, such as dataset segmentation or adaptive gap management policies.

Overall, the data layer provides a standardized preprocessing pipeline that improves data quality and ensures a consistent input representation across different forecasting models while maintaining sufficient flexibility to accommodate diverse IoT datasets.

4.3 Models Considered

Time series forecasting can be approached through a wide variety of modeling techniques. These techniques differ not only in their mathematical formulation but also in their assumptions about the data, computational requirements, and operational characteristics.

Generally speaking, forecasting models can be grouped into different methodological families, including classical statistical models and modern machine learning approaches. Each family provides a different perspective on how temporal dependencies should be represented and exploited for prediction.

The objective of this work is not only to evaluate individual forecasting models but also to analyze how different modeling paradigms behave in a runtime deployment scenario. For this reason, the framework is designed to support heterogeneous models that may require different preprocessing strategies, training procedures, and inference mechanisms.

In the present study, two representative paradigms were selected:

- Classical statistical autoregressive models based on the ARIMA family.
- Deep learning models based on Long Short-Term Memory (LSTM) networks.

These models were chosen because they represent fundamentally different approaches to time series forecasting. **ARIMA models rely on explicitly defined statistical structures**, whereas **LSTM networks learn temporal patterns directly from data** through training.

The following subsections briefly introduce these models and describe their main characteristics.

4.3.1 ARIMA-Based Models

Autoregressive Integrated Moving Average (ARIMA) models are among the most widely used classical approaches for time series forecasting [19]. An ARIMA(p, d, q) model combines three components:

- an autoregressive component of order p, which models the influence of previous observations,

- a moving average component of order q, which models the influence of previous prediction errors.

- a differencing component of order d, which is used to transform non-stationary series into stationary ones,

The autoregressive term captures correlations between the current observation and past values, while the moving average component incorporates information from past forecasting errors. The differencing operator allows the model to handle non-stationary time series by removing trends and stabilizing the mean of the series.

Unlike many modern machine learning approaches, ARIMA models require explicit configuration of their parameters. Selecting appropriate values for (p, d, q) typically involves preliminary analysis of the time series, often using tools such as **autocorrelation and partial autocorrelation functions**.

In addition to the basic ARIMA formulation, extended variants can incorporate additional structures [20]. For example, seasonal ARIMA models (SARIMA) introduce seasonal autoregressive and moving average terms to capture periodic behaviour.

Another common extension consists of enriching the model with Fourier terms used as exogenous regressors. Fourier harmonics allow the model to represent periodic components more effectively, which can be particularly useful when forecasting over longer horizons where standard ARIMA models may converge toward a constant mean.

In the present work, ARIMA models serve as a classical statistical baseline against which more flexible learning-based approaches can be compared.

4.3.2 LSTM Networks

Long Short-Term Memory (LSTM) networks belong to the family of **recurrent neural networks** and are specifically designed to model temporal dependencies in sequential data [21]. Through gated memory mechanisms, LSTMs are able to retain information over long time horizons while mitigating the vanishing gradient problem that affects traditional recurrent architectures.

Unlike ARIMA models, LSTM networks do not rely on an explicitly defined parametric structure. Instead, temporal patterns are learned directly from the training data during the optimization process.

To train an LSTM model, the time series is typically transformed into a sequence of input windows. Each window contains a fixed number of past observations that are used as input to predict one or more future values. The size of this window determines how much historical context the model can use during prediction.

The architecture of the network can vary depending on the design choices made by the developer. Typical configuration parameters include the number of LSTM layers, the

number of hidden units, the use of dropout for regularization, and the size of the input window.

Two main forecasting strategies can be adopted:

- **Single-step forecasting**, where the model is trained to predict only the next observation and longer horizons are obtained recursively by feeding predictions back into the model.

- **Direct multi-step forecasting**, where the network is trained to predict multiple future values simultaneously.

These alternatives provide different trade-offs between model flexibility, training complexity, and forecasting stability over longer horizons [22].

4.3.3 Conceptual Differences Between Modeling Paradigms

ARIMA and LSTM models represent two fundamentally different philosophies for time series forecasting.

ARIMA relies on an explicitly defined statistical structure, where the practitioner must analyze the time series and specify the relevant parameters that describe its temporal behaviour. This approach provides interpretability and computational efficiency but requires **domain knowledge** to configure the model appropriately.

In contrast, LSTM networks adopt a data-driven approach in which temporal dependencies are learned automatically from the training data. Instead of explicitly specifying the lag structure of the model, the practitioner defines the architecture of the neural network and allows the training process to discover relevant patterns.

These differences imply distinct preprocessing requirements, training procedures, and computational characteristics. Consequently, a unified framework capable of handling heterogeneous forecasting models is necessary to enable a fair and systematic comparison between different modeling paradigms.

4.4 Forecasting Framework

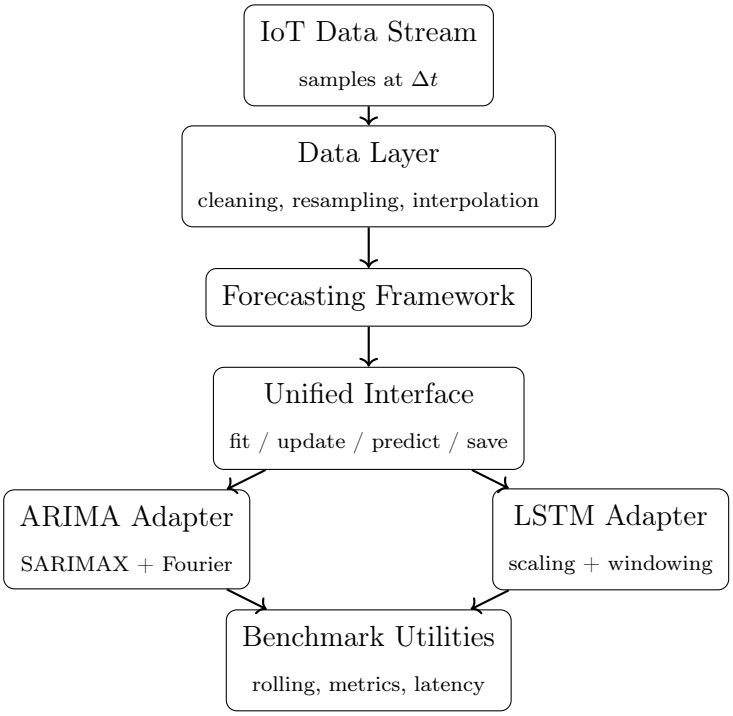

Figure 4.2: High-level architecture of the predictive module within the IoT pipeline.

Forecasting models often differ significantly in their internal structure, preprocessing requirements, and prediction mechanisms. Classical statistical models and neural networks typically operate under different assumptions and require different data representations.

These differences make it difficult to integrate heterogeneous forecasting approaches within a single evaluation pipeline. For this reason, a unified forecasting framework was designed to provide a common interface for training, prediction, and runtime updates while preserving the specific requirements of each model.

The framework abstracts the interaction with forecasting models through a shared interface, while delegating model-specific preprocessing and internal state management to dedicated implementations.

4.4.1 Unified Forecasting Interface

At the core of the framework lies a unified interface that defines the minimal functionality required by a forecasting model operating in a runtime environment.

The interface is implemented in Python in the current prototype; the following listing illustrates its structure in a language-specific form.

```
class RealTimeForecaster:
    def fit(self, series): ...
    def predict(self, H=1): ...
    def update(self, new_data, refit=False): ...
    def copy(self): ...
    def save(self, dirpath): ...
    @classmethod
    def load(cls, dirpath): ...
```

Figure 4.3: Python implementation of the minimal interface adopted by all forecasting models in the framework.

Each forecasting implementation must support the following operations:

- **Training**: fitting the model on a historical time series dataset.
- **Prediction**: generating forecasts for a specified horizon H starting from the most recent observation.
- **Model Update**: incorporating newly observed samples during runtime in order to maintain a consistent forecasting state.
- **Model Persistence**: saving and loading trained models to enable deployment and reuse.

This abstraction allows heterogeneous forecasting models to be integrated within the same pipeline while maintaining a consistent interaction pattern. At the same time, the framework allows each implementation to internally manage model-specific preprocessing steps and state variables.

4.4.2 Model Implementations

Concrete forecasting models are implemented through adapters that conform to the unified interface while encapsulating the internal logic required by each algorithm.

These adapters handle model configuration, preprocessing operations, and the management of internal state variables necessary for runtime prediction.

ARIMA Adapter

The ARIMA adapter provides a wrapper around a statistical forecasting implementation based on the SARIMAX model.

The adapter exposes configuration parameters that allow specifying the autoregressive order, differencing order, and moving average order that define the ARIMA structure.

Optional seasonal parameters can also be provided when necessary. In addition, the implementation supports the inclusion of Fourier harmonics as exogenous regressors in order to better capture periodic components of the time series.

Compared to other forecasting approaches, ARIMA models align naturally with the unified interface since they directly support multi-step forecasting without explicit recursive prediction loops. The behaviour is as follows: once the model is fitted, it is possible to request a forecast for an arbitrary number of future steps without explicitly feeding predicted values back into the model.

This behaviour simplifies the implementation of the prediction method, as the underlying ARIMA model internally handles the recursive forecasting process.

The adapter also provides support for updating the model as new observations become available. Inspired by the refitting mechanisms available in SARIMAX implementations, the update operation allows optionally refitting the model in order to incorporate new information during runtime.

From a preprocessing perspective, ARIMA models introduce a different set of requirements compared to neural approaches. In particular, the model assumes that the input time series is stationary. For this reason, the implementation provides utilities that assist in verifying and transforming the dataset in order to satisfy stationarity conditions before training.

This design reflects a key principle of the framework: while all models receive the same standardized dataset as input, each adapter is responsible for performing any additional preprocessing required by the specific forecasting algorithm.

LSTM Adapter

The LSTM adapter implements forecasting models based on recurrent neural networks.

Unlike ARIMA models, LSTM networks require a more complex data preparation process before training and inference can be performed. Neural networks typically expect normalized input data, since large differences in value scale can negatively affect the optimization process. For this reason, the adapter internally manages data scaling operations and automatically converts predictions back to the original scale when returning results.

In addition to normalization, LSTM models require transforming the time series into sliding windows of fixed length. Each window represents a sequence of past observations that serves as input for predicting future values.

The architecture of the neural network is defined through a configurable building function passed to the adapter during initialization. This design allows different network structures to be experimented with, avoiding the need to modify the surrounding framework. Parameters controlling the model architecture and training process are passed through configuration dictionaries, allowing flexible experimentation with different hyperparameter combinations.

During runtime operation, the adapter maintains several internal state variables required for prediction. These include the scaler used during training, the current input window representing the most recent observations, and the parameters defining the network architecture.

Maintaining this internal state is necessary in order to correctly update the input sequence as new observations or predicted values become available during forecasting.

4.4.3 Benchmarking Utilities

In addition to model abstraction, the framework provides a set of utilities designed to support systematic evaluation of forecasting models.

A key component of this evaluation process is the simulation of a rolling forecasting scenario. In this setting, the model generates predictions using only the information available up to a given time instant, reproducing the behaviour of a real-time deployment environment.

Starting from a trained model, the evaluation procedure repeatedly performs the following steps:

1. generate a forecast for a specified horizon H,
2. compare the predicted values with the corresponding ground truth observations,
3. update the model with the newly observed sample.

This procedure simulates an online forecasting scenario in which predictions must be produced without access to future information.

When applied to an entire test dataset, the rolling evaluation may require a very large number of predictions, especially for long time series or computationally expensive models. For this reason, the framework allows limiting the number of evaluation points.

Instead of evaluating predictions at every time step, the user can specify a number of evaluation points that are distributed uniformly across the test interval. This strategy preserves a representative evaluation of the forecasting behaviour while significantly reducing the computational cost of the simulation.

4.5 Evaluation Methodology

The objective of the evaluation phase is to assess forecasting models under both predictive accuracy and runtime feasibility constraints, reflecting the operational requirements introduced in Section 4.1.1.

While the previous sections presented the forecasting models and the supporting framework, this section defines the experimental protocol used to evaluate their performance.

The methodology focuses on how predictions are generated and measured, independently from the specific numerical results, which are presented in Chapter 5.

4.5.1 Rolling Forecast Protocol

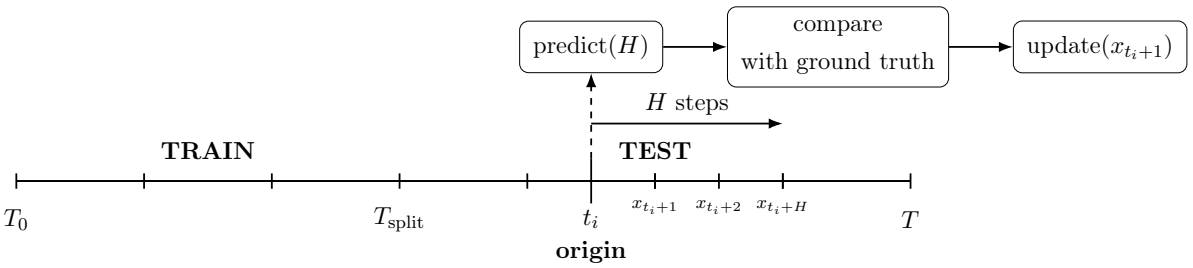

Figure 4.4: Rolling multi-origin evaluation protocol. At each selected origin t_i, the model predicts H steps ahead, predictions are evaluated against ground truth, and the model state is updated with newly observed samples.

To reproduce the behaviour of a real deployment scenario, forecasting models are evaluated using a rolling multi-origin forecasting procedure.

After training on an initial historical segment, the model is applied sequentially along the test portion of the dataset. At selected forecast origins, predictions are generated using only the observations available up to that point in time.

Formally, given a test series $\{x_t\}_{t=T_0}^{T}$ and a forecast horizon H, forecasts are produced at multiple time origins t_i such that the ground truth observations up to $t_i + H$ are available for evaluation.

This procedure simulates an online deployment scenario in which predictions must be produced without access to future information.

4.5.2 Multi-Horizon Evaluation

Forecasting performance is evaluated across multiple prediction horizons in order to analyze how model accuracy evolves as the forecast distance increases.

Short horizons (e.g., $H = 1$) capture the model's ability to accurately predict the immediate next observation, corresponding to the strictest real-time requirement.

Longer horizons provide insight into the robustness of the forecasting model when predictions must cover extended periods without access to new observations.

In IoT environments, the duration of potential sensor outages or communication disruptions is often uncertain and strongly dependent on the specific application context. Evaluating multiple horizons therefore allows the analysis to cover different operational scenarios, ranging from short interruptions to longer periods in which predicted values may temporarily replace missing measurements.

Moreover, prediction errors typically accumulate as the forecast horizon increases, making long-horizon evaluation particularly useful for assessing the structural stability of forecasting models.

4.5.3 Accuracy Metrics

Forecast accuracy is assessed by comparing predicted values with the corresponding ground-truth observations across all forecast origins.

For each forecast origin, the error is computed over the H predicted steps. The final score is obtained by aggregating the error across all forecast origins.

Formally, let x_{t_i+h} be the true value and $\hat{x}_{t_i+h}$ the prediction produced at forecast origin t_i for horizon h. The RMSE for a single origin is:

$$RMSE_i = \sqrt{\frac{1}{H}\sum_{h=1}^{H}(x_{t_i+h} - \hat{x}_{t_i+h})^2}$$

The final reported score is the mean across all forecast origins:

$$RMSE = \frac{1}{M}\sum_{i=1}^{M} RMSE_i$$

where M denotes the number of evaluated forecast origins.

RMSE is widely used in time series forecasting because it penalizes large prediction errors more strongly than linear error metrics.

While additional metrics such as Mean Absolute Error (MAE) or Mean Absolute Percentage Error (MAPE) may provide complementary perspectives, RMSE offers a clear and interpretable measure of the overall forecasting deviation and is therefore used as the primary indicator of predictive performance.

4.5.4 Latency Measurement

In addition to predictive accuracy, runtime feasibility is assessed by measuring the latency required to generate forecasts.

For each forecast call, the time required to compute an H-step prediction is measured using high-resolution timing functions.

Latency statistics include:

- mean inference time,
- 95th percentile latency,
- maximum observed latency.

Latency is measured using wall-clock time through the `time.perf_counter()` function in Python.

Two latency components are evaluated:

- **Update latency**, measuring the time required to integrate newly observed data into the model state through the `update()` operation.
- **Forecast latency**, measuring the time required to produce an H-step prediction through the `predict(H)` method.

Both measurements are collected for each evaluated forecast origin during the rolling forecasting simulation.

Latency values are expressed in milliseconds and represent wall-clock execution time measured during the simulation.

However, in a real-time deployment scenario the forecasting loop typically involves both updating the model with newly observed data and producing predictions. The effective runtime constraint can therefore be expressed as

$$T_{update} + T_{forecast} \leq \Delta t$$

where Δt represents the sampling interval of the monitored system.

Both prediction and update latencies are measured explicitly within the rolling simulation, since in a real-time deployment both operations contribute to the overall loop responsiveness.

4.5.5 Runtime Considerations and Forecast Scheduling

The relationship between prediction latency and model update cost also influences the choice of forecasting strategy.

In some scenarios it may be beneficial to generate multiple predictions at once rather than producing a single prediction at every sampling instant. This approach effectively reduces the frequency at which the model must be invoked.

However, this strategy is advantageous only if generating a multi-step forecast is computationally cheaper than executing the same number of single-step predictions sequentially.

Consequently, the evaluation also provides insight into whether forecasting models benefit from batch prediction strategies or whether single-step streaming predictions remain preferable.

The experimental protocol defined in this chapter establishes the methodology used to analyze forecasting behaviour under realistic operational constraints.

The following chapter applies this protocol to the considered models and datasets, presenting the experimental results and discussing the observed trade-offs between forecasting accuracy, latency, and runtime feasibility.

Chapter 5

System Evaluation

This chapter presents the experimental evaluation of the IoT framework introduced in Chapter 3. The goal of the evaluation is twofold. First, we analyse the behaviour and scalability of the data ingestion pipeline under increasing workload and progressively improved message routing strategies. Second, we assess the performance of the predictive module described in Chapter 4, with particular attention to runtime latency and the trade–off between forecast accuracy and computational cost. The results reported here are the outcome of a systematic testing process that explores several architectural variations, quantifies their impact on end–to–end latency and throughput, and demonstrates how incremental refinements can significantly improve system performance.

The chapter is organised into two main sections. In Section 5.2 we present the scalability tests carried out on the IoT pipeline. Starting from a baseline design where all measurements are published on a single topic, we introduce progressive improvements in the topic hierarchy and subscriber implementation. Each subsection introduces a new configuration, summarises the measured variables, and discusses the observed behaviour. In Section 5.3 we shift the focus to the predictive module, presenting results from the forecasting models evaluated under real–time constraints and discussing the criteria used for runtime–aware model selection.

5.1 Experimental Setup

The experiments were performed on a dataset of environmental measurements collected from an IoT deployment. Sensors report temperature, humidity and other metrics at a configurable sampling interval. For the pipeline scalability tests we preserved the native high–frequency sampling of one second in order to stress the communication and processing pipeline. For the predictive model evaluation the data were resampled to a one–minute interval because the dynamics of the monitored variables evolve slowly and minute–resolution forecasts are sufficient for most control actions.

All experiments were executed on a server equipped with an Intel Xeon W – 2123

CPU running at 3.60 GHz (4 cores) and 43 GB of RAM. The persistence layer used a time–series database hosted on a network–mounted filesystem with a measured sequential write throughput of approximately 109 MB/s. Sensors published to a serverless EMQX MQTT broker, and a subscriber component running on the same host consumed the messages and stored them in the database.

To simulate different workload conditions we generated synthetic streams from multiple virtual sensors. Each virtual sensor published periodic messages with a fixed payload structure. The number of sensors L and the sampling frequency R were varied in order to explore different traffic regimes. For each configuration we measured the following system–level metrics:

- **Queue latency**: the waiting time incurred by a message in the subscriber queue before processing.
- **Processing latency**: the time required to persist a single message to the database.
- **End–to–end latency**: the sum of queue and processing latencies, measured from the moment a message is published until it is stored.
- **Throughput**: the number of messages successfully processed per second.
- **Loss rate**: the fraction of messages that were not processed before the experiment completed. Lost messages result from queue overflows when the arrival rate exceeds the processing capacity.

Each experiment lasted two minutes per sensor (120 samples) so that all runs could be compared over equal durations. Unless otherwise specified, the per–sensor sampling interval was one second.

5.2 Scalability of the IoT Data Ingestion Pipeline

The scalability of the data ingestion pipeline was evaluated through a progressive refinement of the system architecture. Starting from a minimal baseline configuration, each experiment introduces a design modification aimed at addressing a specific bottleneck observed in the previous setup.

The evaluation is conducted in the context of a smart environment composed of multiple rooms equipped with environmental sensors measuring temperature, humidity, and CO_2 levels. Sensors periodically publish measurements to an MQTT broker, while a subscriber component consumes the incoming messages and persists them into a time-series database.

The goal of this evaluation is to analyze how different architectural configurations affect the ability of the system to process incoming measurements as the workload increases. In

particular, the experiments focus on identifying the main bottlenecks in the message processing pipeline and evaluating how different design choices improve the overall throughput and latency of the system.

Each subsection introduces a specific architectural refinement and evaluates its impact under increasing workload conditions.

5.2.1 Baseline: Single Topic Subscriber

To establish a reference point for the evaluation of the ingestion pipeline, the simplest possible communication architecture is first considered.

In this baseline configuration, all sensor measurements are published to a single MQTT topic and processed by a single subscriber. This represents the most straightforward organization of the communication layer, where the entire sensing infrastructure communicates through a single channel.

As a consequence, the subscriber receives measurements from all monitored rooms and environmental variables, including temperature, humidity, and CO_2. Every incoming message must therefore be processed sequentially by the same subscriber instance and persisted to the database.

Although this configuration is intentionally simple, it provides a useful reference point for understanding how the ingestion pipeline behaves under increasing workload. In particular, this setup makes it possible to observe how the system reacts as the number of active sensors grows.

To analyze the system behavior, two latency components are measured: the delivery latency and the processing latency. The delivery latency represents the time elapsed between the publication of a message and its reception by the subscriber, while the processing latency captures the time required by the subscriber to process and store the message after it has been received.

Figure 5.1 reports the delivery latency observed as the number of sensors increases. The metric shown in the figure corresponds to the 95th percentile (p95), which represents the latency value below which 95% of the observations fall. This metric is commonly used to characterize worst-case system behavior while filtering out extreme outliers.

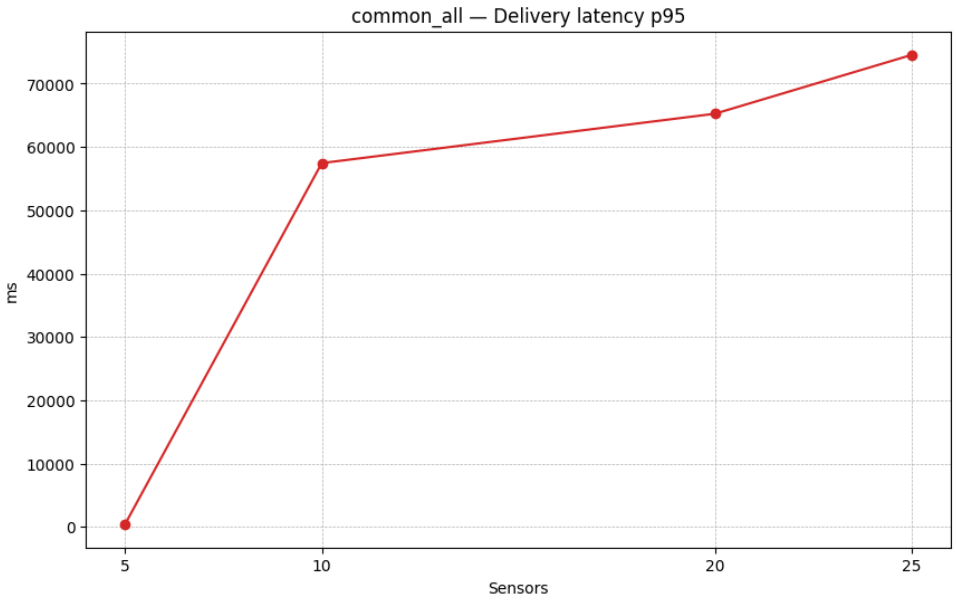

Figure 5.1: Delivery latency (p95) in the baseline single-topic configuration as the number of sensors increases.

As shown in the figure, the delivery latency rapidly increases as the number of sensors grows. When the system is evaluated with five sensors, the latency remains relatively low, but it increases sharply once the number of sensors reaches ten and continues to grow as the workload increases further.

To better understand the origin of this latency growth, the internal processing cost of the subscriber is analyzed separately. The processing latency measured inside the subscriber is reported in Figure 5.2.

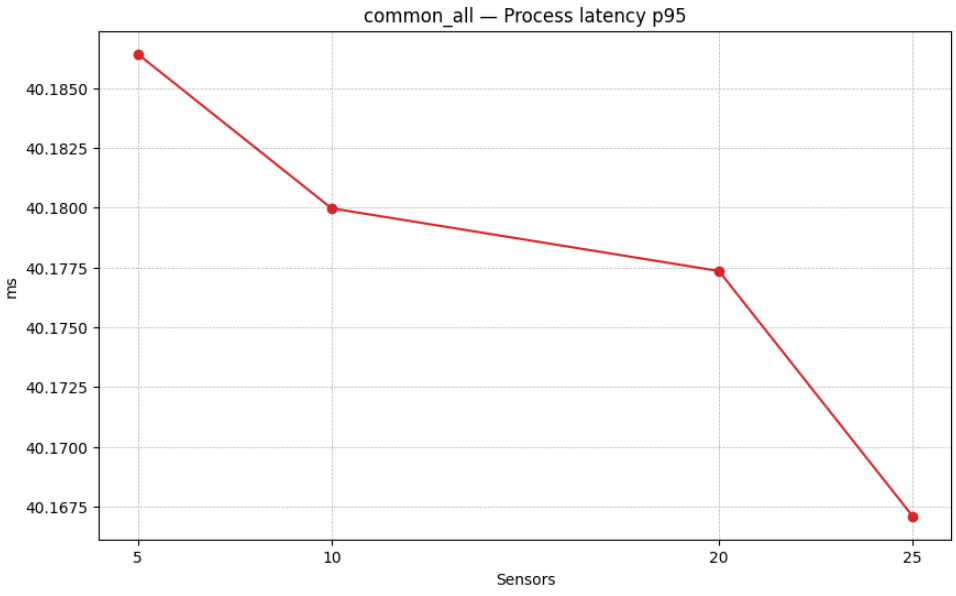

Figure 5.2: Subscriber processing latency in the baseline configuration.

Interestingly, the processing latency remains almost constant across all configurations, with an average cost of approximately 40 ms per message. This indicates that the computational cost of handling individual messages does not significantly increase with the number of sensors.

The divergence between delivery latency and processing latency suggests that the observed delay does not originate from the processing stage itself, but rather from the accumulation of messages before they are processed.

Given a processing time of approximately 40 ms per message, the maximum sustainable throughput of the subscriber is roughly 25 messages per second. When the incoming message rate approaches or exceeds this threshold, messages begin to accumulate within the ingestion pipeline, leading to a rapid increase in end-to-end latency.

The impact of this behavior is also visible in terms of message reliability. Figure 5.3 reports the fraction of messages that are not successfully processed as the number of sensors increases.

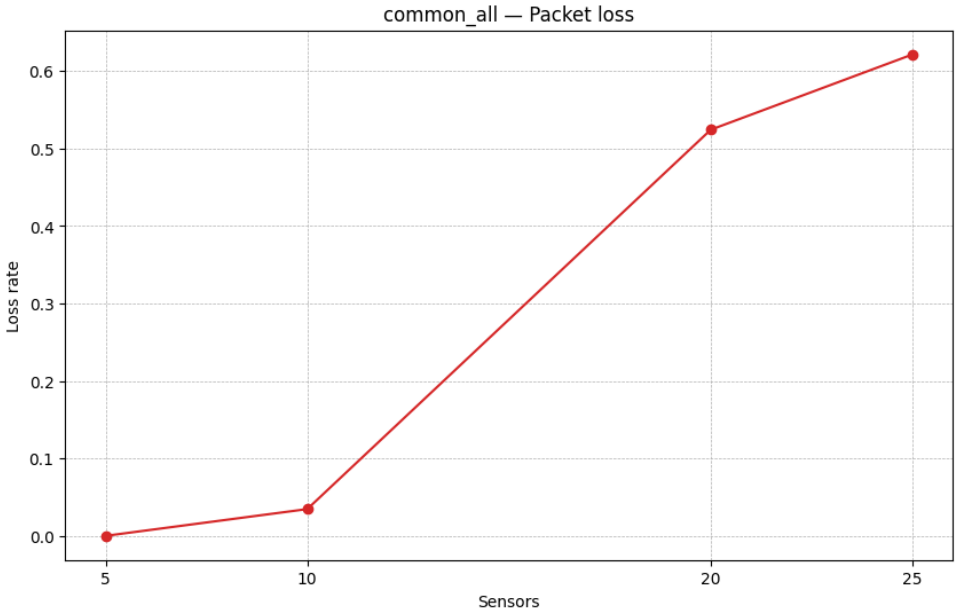

Figure 5.3: Message loss rate observed in the baseline configuration as the number of sensors increases.

The results show that **message loss remains negligible** when the number of active sensors is limited. However, as the workload increases, the fraction of lost messages grows rapidly. In the most demanding configurations, a significant portion of the generated measurements fails to be processed.

These results indicate that the ingestion pipeline becomes unable to sustain the incoming data rate once the workload exceeds a certain threshold. Messages accumulate within the communication pipeline and a portion of them is not successfully processed.

However, these observations alone are not sufficient to determine whether the bottleneck

originates from the subscriber processing stage or from other components of the pipeline, such as message publication or delivery behavior. For this reason, additional experiments are conducted in the following sections to further isolate the source of the observed performance degradation.

5.2.2 Subscriber-Side Metric Filtering

A first improvement over the baseline configuration consists of reducing the number of persistence operations performed by the subscriber.

In this configuration, all sensor measurements continue to be published to the same MQTT topic, preserving the communication structure of the baseline architecture. However, the subscriber is configured to persist only a specific metric (e.g., temperature) while discarding the remaining ones.

As a consequence, the subscriber still receives the complete stream of messages generated by the sensing infrastructure, but only a subset of them is written to the database. This modification reduces the amount of storage operations without altering the communication pattern of the system.

To evaluate the impact of this change, the same experimental setup used in the baseline configuration is repeated while limiting persistence to a single metric.

Figure 5.4 reports the delivery latency observed under this configuration.

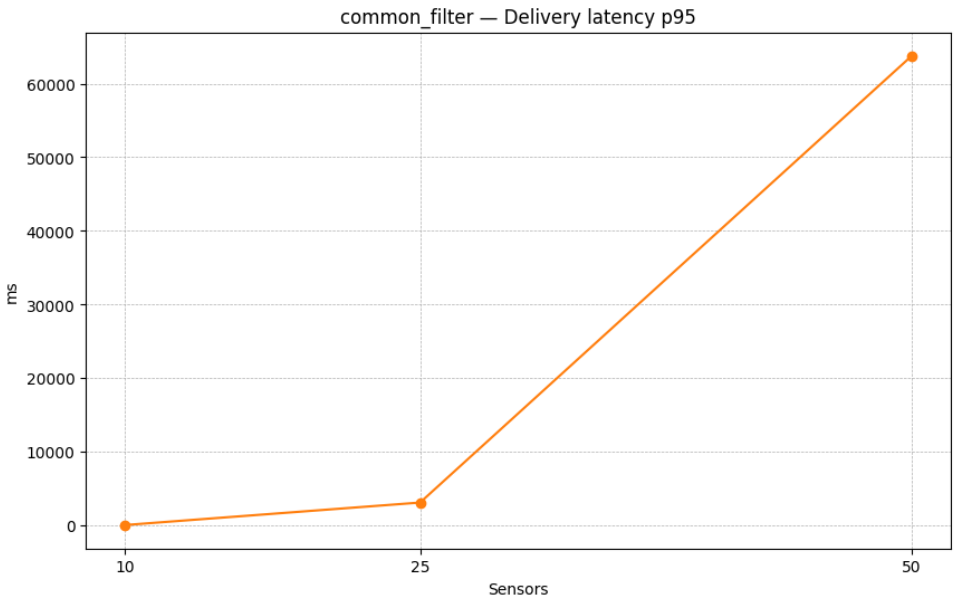

Figure 5.4: Delivery latency (p95) with subscriber-side metric filtering.

Compared to the baseline configuration, the system exhibits a significant reduction in latency under moderate workloads. When the system is evaluated with ten sensors, the delivery latency remains below 100 ms, which represents a substantial improvement over the baseline scenario.

As the number of sensors increases, however, the delivery latency begins to grow again. With twenty-five sensors the latency reaches several seconds, and with fifty sensors it exceeds tens of seconds, indicating that the system once again approaches saturation.

The impact of this behavior can also be observed in terms of message reliability. Figure 5.5 reports the fraction of messages that are not successfully processed as the workload increases.

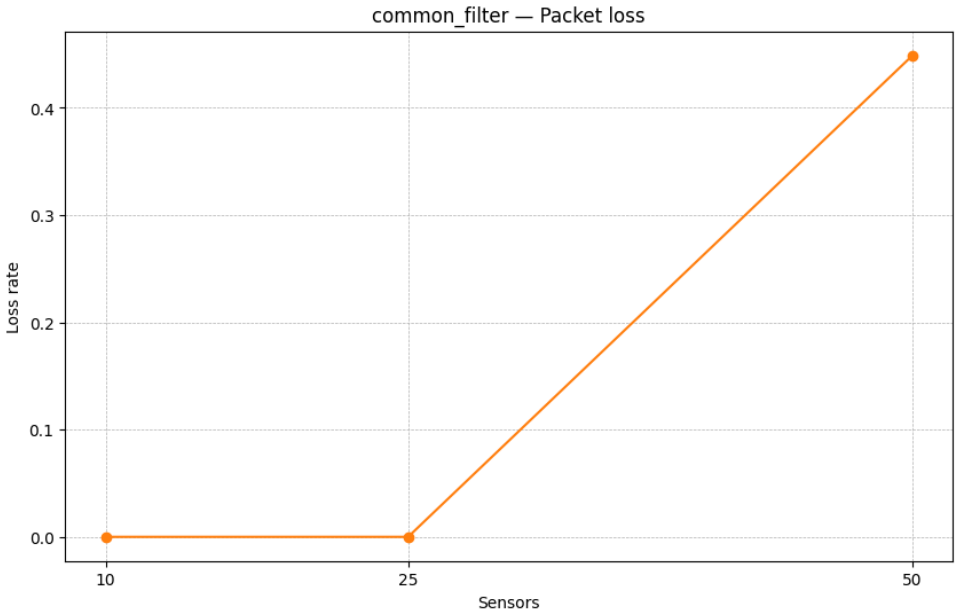

Figure 5.5: Packet loss observed with subscriber-side metric filtering.

The results show that **message loss remains negligible** under moderate workloads, with no observed losses for configurations up to twenty-five sensors. However, when the system is evaluated with fifty sensors, the loss rate increases significantly, indicating that the system again becomes unable to sustain the incoming message rate.

Although this configuration substantially reduces the number of database persistence operations, it does not eliminate the fundamental scalability limitation of the baseline architecture. Since all measurements are still published on the same MQTT topic, the subscriber must continue to receive and inspect every incoming message before deciding whether it should be stored.

As a consequence, the amount of traffic delivered to the subscriber remains unchanged. The filtering operation only reduces the persistence workload, but does not reduce the communication load itself.

This observation suggests that further improvements require reducing the amount of traffic delivered to the subscriber at the MQTT level, rather than filtering messages only after they have already been received.

5.2.3 Metric-Based Topic Organization

A further improvement over the previous configuration consists of moving the filtering mechanism upstream to the MQTT broker.

Instead of publishing all measurements to a single shared topic, measurements are organized according to the monitored variable. Each metric is therefore associated with a dedicated topic. For instance, temperature measurements are published to a specific temperature topic, while humidity and CO_2 measurements are delivered through separate channels.

With this organization, subscribers interested in a particular variable can subscribe only to the corresponding topic. As a consequence, irrelevant measurements are no longer delivered to the subscriber, reducing the amount of traffic that must be inspected and processed.

Figure 5.6 reports the delivery latency observed under this configuration when the subscriber receives only temperature measurements.

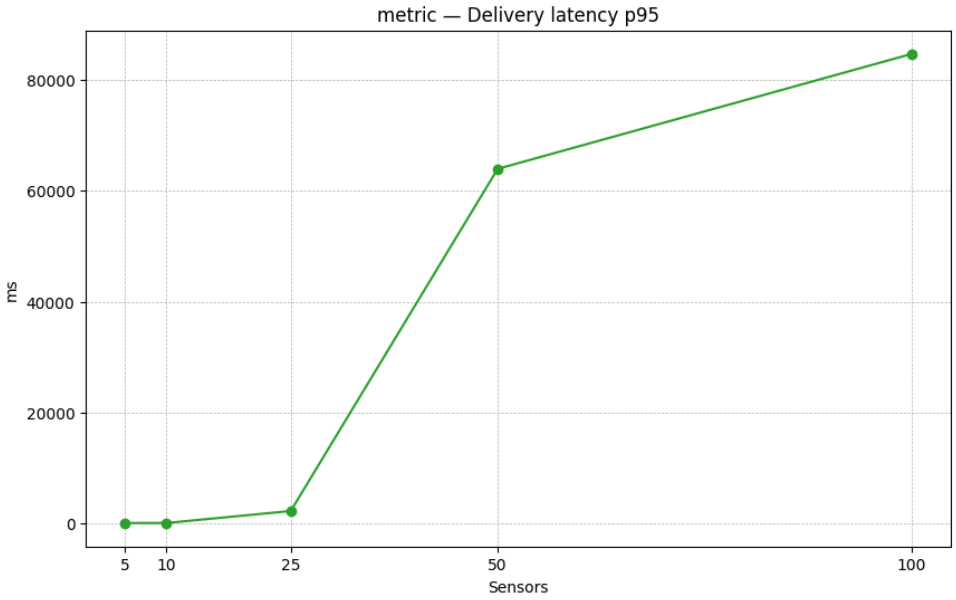

Figure 5.6: Delivery latency (p95) with metric-based topic organization.

The results show that latency remains very low under moderate workloads. With up to ten sensors, the delivery latency remains below 150 ms, indicating that the system can comfortably sustain the incoming message rate.

However, as the number of sensors increases, the system again approaches saturation. When evaluated with twenty-five sensors, the delivery latency rises to several seconds, and for larger workloads it increases dramatically, exceeding tens of seconds.

To better understand the impact of metric-based topic organization, Figure 5.7 compares the delivery latency observed in this configuration with the one previously measured when filtering measurements directly at the subscriber.

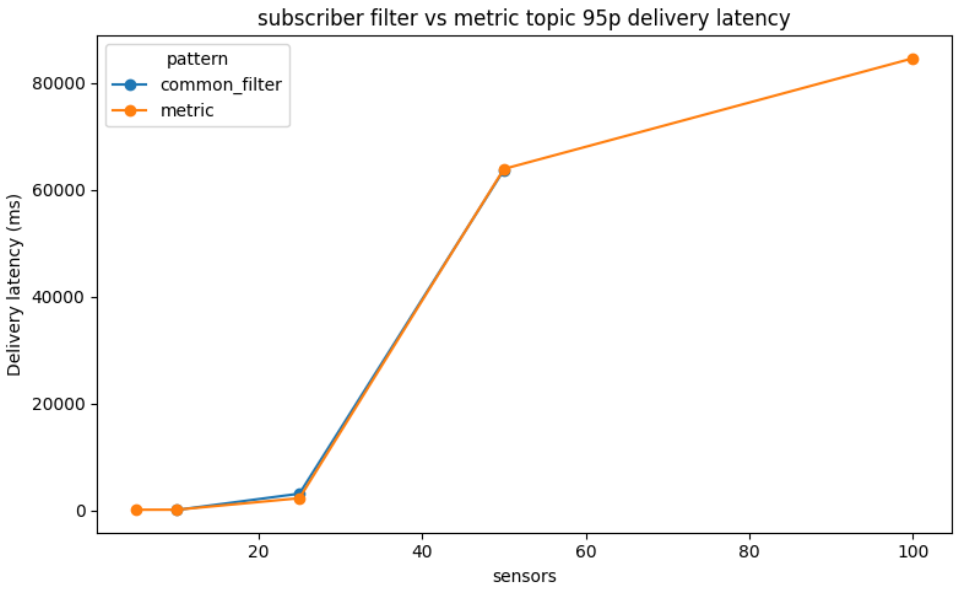

Figure 5.7: Comparison between subscriber-side filtering and metric-based topic organization.

As shown in the figure, the two configurations exhibit very similar behavior across all workloads. The delivery latency curves almost overlap, indicating that organizing topics by metric does not produce a substantial improvement compared to filtering measurements directly at the subscriber.

This result suggests that the filtering operation itself is not the primary factor affecting the scalability of the system. In both configurations the subscriber ultimately processes a comparable number of relevant measurements, and the cost of message parsing and filtering remains relatively small.

As in the previous experiments, the internal processing latency remains approximately constant at around 40 ms per message, confirming that the computational overhead associated with message handling is limited.

Consequently, while metric-based topic organization reduces the delivery of irrelevant measurements, it does not eliminate the fundamental scalability limitations observed in the previous configurations.

In particular, the system remains highly sensitive to the number of active sensors. As the sensing infrastructure grows, the overall message rate increases proportionally, and the single subscriber instance becomes responsible for processing a continuously larger portion of the incoming traffic. This behavior reveals a clear limitation in terms of horizontal scalability, since the ingestion architecture does not yet distribute the workload across multiple independent processing paths.

These observations suggest that further improvements require a stronger partitioning of the data stream, so that each subscriber receives only a small fraction of the overall sensing traffic. For this reason, the following section evaluates a hierarchical topic organization

that partitions measurements according to both location and metric.

5.2.4 Hierarchical Topic Design

The metric-based topic organization introduced in the previous configuration reduces the delivery of irrelevant measurements by separating data streams according to the monitored variable. However, each subscriber responsible for a given metric still receives measurements generated by all sensors in the monitored environment.

As a consequence, the workload of the subscriber continues to grow proportionally with the number of sensors deployed in the system. This behavior limits the horizontal scalability of the ingestion pipeline, since a single subscriber remains responsible for processing an increasingly large stream of measurements.

To address this limitation, the topic structure can be extended using a hierarchical organization that encodes both the location and the metric within the topic name.

In this configuration, each measurement is published to a topic that includes both the room identifier and the monitored variable, for instance:

```
project/room_id/temperature
```

This organization allows subscribers to select highly specific subsets of the data stream. For example, a subscriber responsible for storing temperature measurements for a particular room can subscribe only to the corresponding topic. By partitioning the data stream according to both location and metric, the workload can be naturally distributed across multiple subscribers.

Figure 5.8 reports the delivery latency observed when a subscriber processes temperature measurements from a single room while the total number of sensors in the system increases.

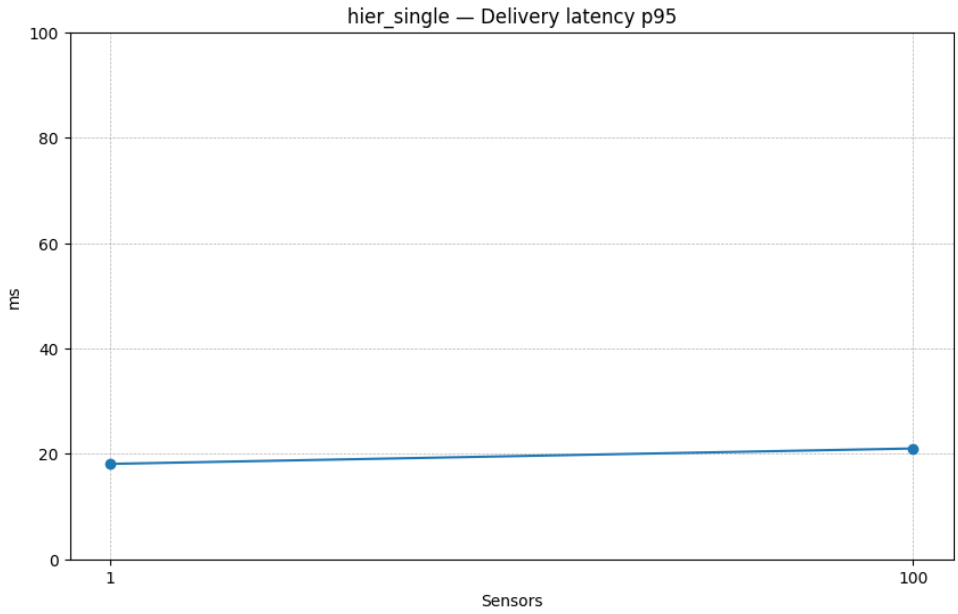

Figure 5.8: Delivery latency under hierarchical topic organization.

Unlike the previous configurations, the latency remains almost constant even when the total number of sensors increases significantly. This behavior indicates that the workload of the subscriber is no longer coupled to the global size of the sensing infrastructure.

In this configuration, the subscriber receives only the measurements associated with a specific location and metric. As a consequence, the incoming message rate remains stable regardless of how many additional sensors are deployed elsewhere in the system.

This result demonstrates that hierarchical topic organization enables effective horizontal scalability of the ingestion pipeline by partitioning the global data stream into smaller, independent processing paths.

Although this routing strategy significantly improves scalability, each subscriber still processes messages sequentially. When the message generation rate of an individual sensor increases beyond the processing capacity of the subscriber, messages may still accumulate within the ingestion pipeline.

For this reason, the following experiment investigates the impact of higher message frequencies on the subscriber architecture.

5.2.5 Impact of Message Rate on the Subscriber

The hierarchical topic organization discussed in the previous section successfully decouples the workload of each subscriber from the total number of sensors deployed in the system. As a consequence, the ingestion pipeline achieves good horizontal scalability with respect to the size of the sensing infrastructure.

However, this improvement does not automatically guarantee robustness with respect to the message generation rate. Even when each subscriber is responsible for only a single

location and metric, the incoming stream may still become too frequent for the ingestion pipeline to handle efficiently.

To investigate this aspect, additional experiments were performed by keeping the hierarchical topic organization fixed while increasing the message rate generated by a single sensor. In this phase, the objective was to evaluate whether the subscriber architecture remained stable as the publication frequency increased.

Figure 5.9 reports the delivery latency observed with the direct subscriber configuration, while Figure 5.10 shows the corresponding message loss rate.

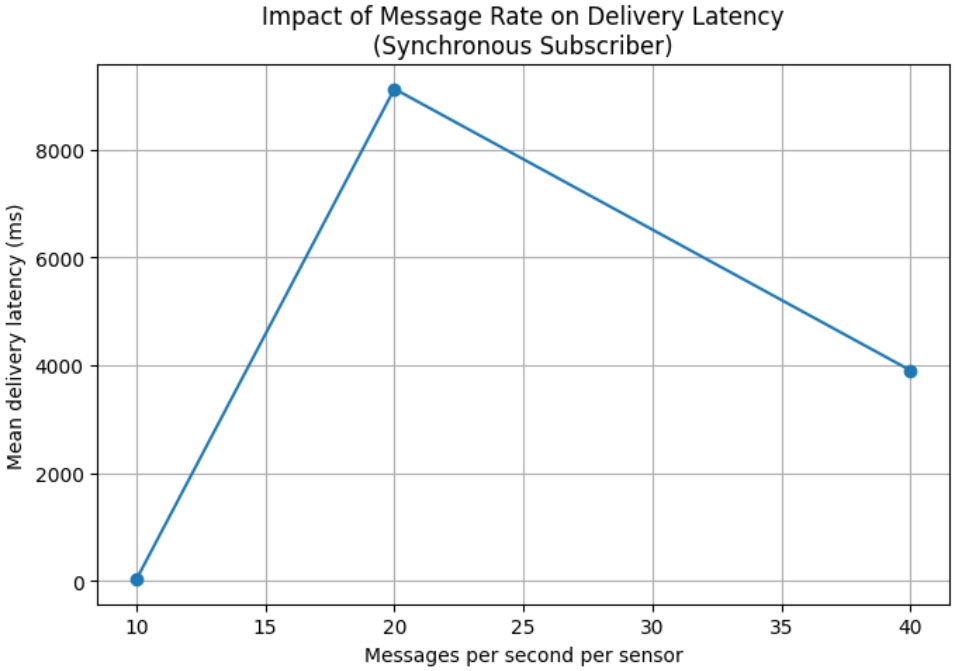

Figure 5.9: Delivery latency under increasing message rates with direct subscriber processing.

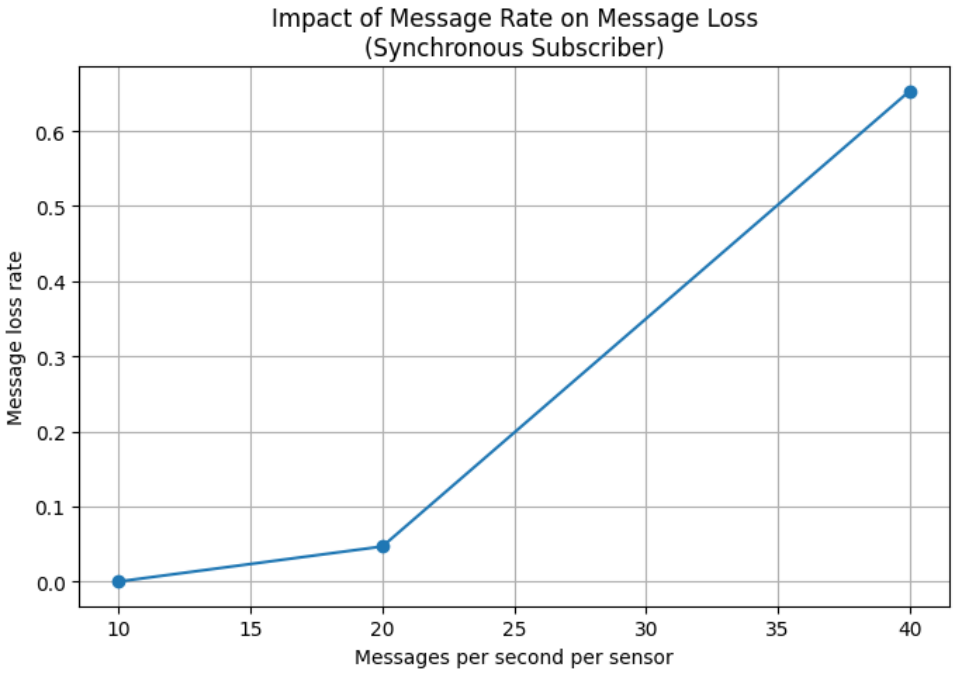

Figure 5.10: Message loss rate under increasing message rates with direct subscriber processing.

The results show that the system remains stable at moderate message rates. With a generation rate of 10 messages per second, all messages are successfully processed and the delivery latency remains limited.

When the message rate is increased to 20 messages per second, a clear degradation appears. Although the number of sensors remains unchanged, the delivery latency rises sharply and message loss begins to occur. When the rate is further increased to 40 messages per second, the loss rate exceeds 65%, indicating that the ingestion pipeline is no longer able to sustain the incoming stream.

These results show that, although hierarchical routing solves the scalability problem with respect to the number of sensors, the system remains highly sensitive to the frequency of the incoming measurements. Therefore, the next step consists of evaluating whether concurrent processing at the subscriber side can mitigate this limitation.

5.2.6 Concurrent Subscriber with Thread Pool

To investigate whether the degradation observed at high message rates originates from the sequential processing model of the subscriber, the architecture was extended with a thread pool.

In this configuration, incoming messages are first received by the MQTT callback and then delegated to a pool of worker threads responsible for their processing. In principle, this design should reduce waiting times inside the subscriber and improve the ability of

the system to sustain higher message rates.

Figure 5.11 reports the delivery latency observed with a thread pool of four workers, while Figure 5.12 shows the corresponding message loss rate.

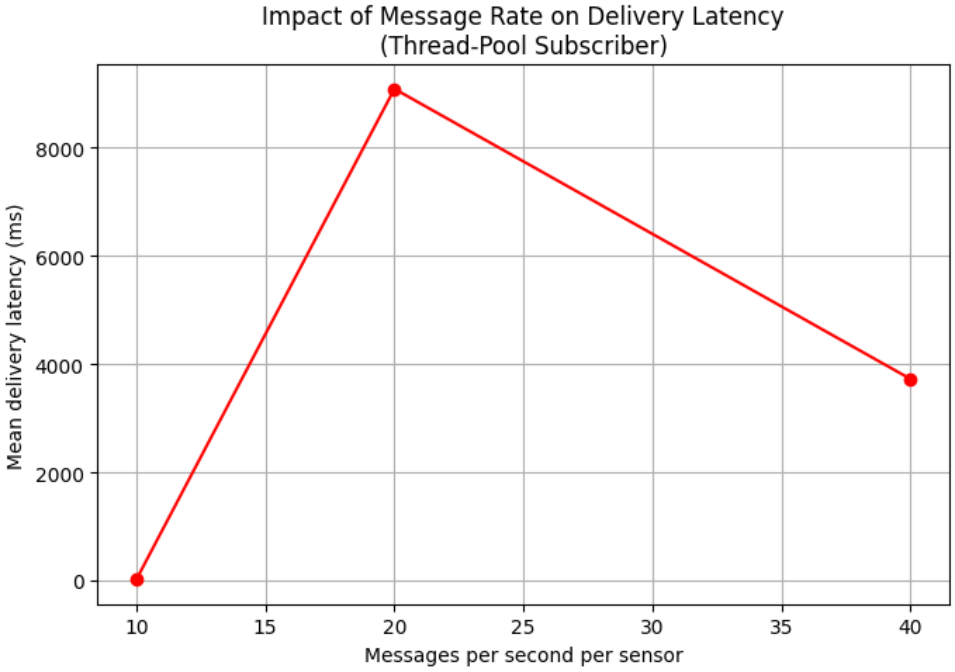

Figure 5.11: Delivery latency under increasing message rates with a thread-pool subscriber.

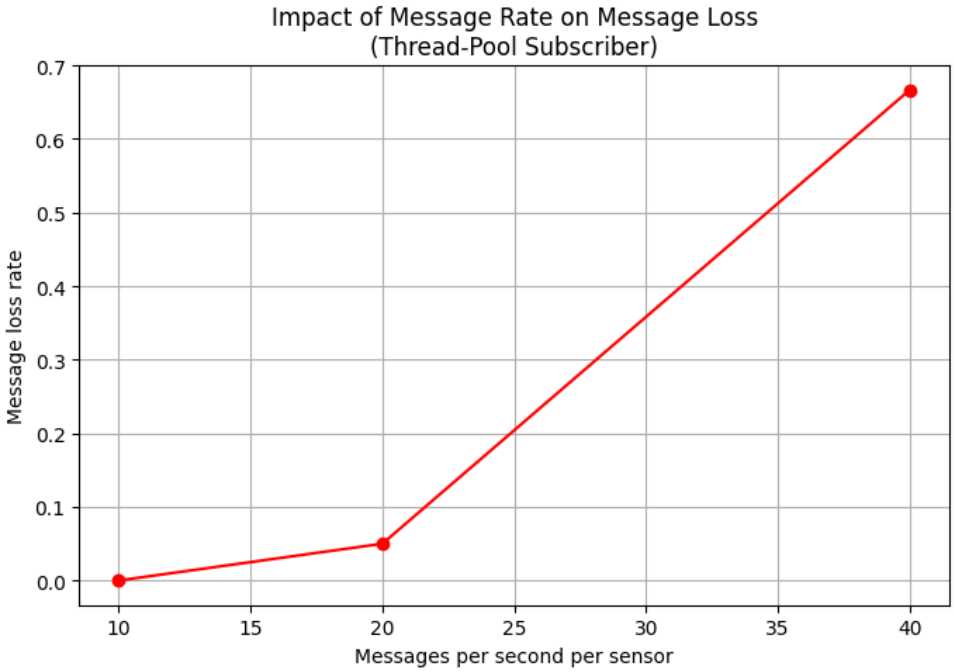

Figure 5.12: Message loss rate under increasing message rates with a thread-pool subscriber.

To directly compare the two subscriber architectures, Figure 5.13 contrasts the delivery

latency of the direct subscriber and the thread-pool subscriber as the message rate increases.

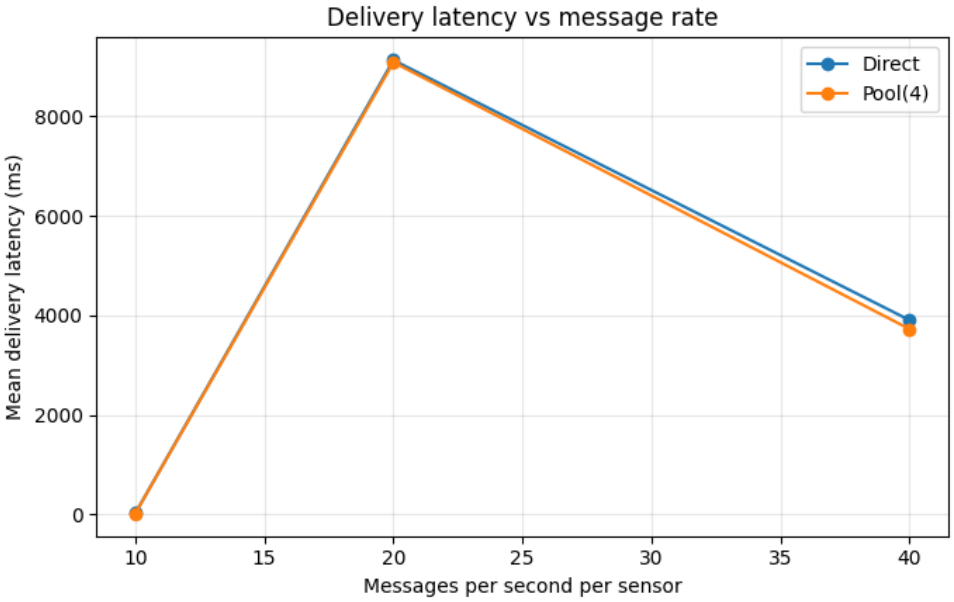

Figure 5.13: Comparison between direct and thread-pool subscriber configurations under increasing message rates.

Contrary to the initial expectation, the concurrent architecture does not produce a substantial improvement. The delivery latency and message loss observed with the thread pool remain very similar to those measured with direct processing. At 20 messages per second, both configurations already exhibit severe latency growth; at 40 messages per second, both configurations collapse with comparable loss rates.

This result suggests that the dominant limitation is not located in the internal processing stage of the subscriber. If the subscriber were the main bottleneck, a clearer improvement would be expected after introducing concurrent workers. Since this does not occur, the source of the degradation must be sought in other components of the ingestion pipeline.

For this reason, an additional investigation was conducted on the publication side in order to determine whether the observed losses and latency growth were introduced before the messages reached the subscriber. As discussed in the following section, this analysis showed that the dominant bottleneck under high-frequency conditions was located at the publisher side rather than in the subscriber architecture itself.

5.2.7 Publisher-Side Message Loss

The previous experiments highlighted a clear degradation of the system when the message generation rate increases. In particular, increasing the publication frequency leads to a rapid growth of delivery latency and to a significant number of lost messages.

Initially, this behavior was attributed to the processing model of the subscriber. Since

the synchronous subscriber processes messages sequentially, it was hypothesized that the observed losses could be caused by the inability of the subscriber to keep up with the incoming message stream.

To test this hypothesis, the subscriber architecture was extended with a thread pool that allows multiple messages to be processed concurrently. However, the experimental results showed that the concurrent subscriber does not significantly improve either latency or message loss compared to the direct processing configuration. The performance of the two architectures remains very similar even when the message rate increases.

This observation suggests that the dominant limitation may not be located in the subscriber processing stage. If the subscriber were the main bottleneck, a clear improvement would be expected when introducing concurrent workers.

For this reason, the investigation was extended to the publication stage. In particular, the actual message publication rate was measured and compared with the target frequency requested by the experiment.

Figure 5.14 reports the effective publication rate observed as the target frequency increases.

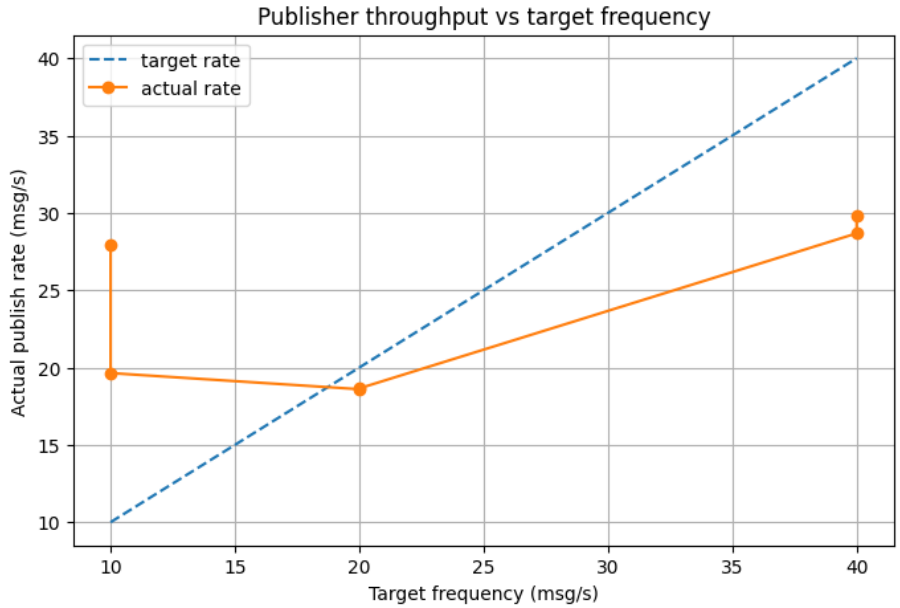

Figure 5.14: Publisher throughput compared with the target message generation rate.

The results show that the publisher is unable to sustain the requested transmission frequency. While the target rate increases up to 40 messages per second, the effective publication rate saturates at significantly lower values. This behavior indicates that, under high-frequency conditions, the publisher itself becomes a bottleneck.

As a consequence, some measurements are effectively dropped before reaching the MQTT broker. This explains the message loss observed in the previous experiments, which does not originate from the subscriber processing stage but rather from the message

generation process.

This phenomenon highlights an important characteristic of many IoT monitoring systems. In practical deployments, perfect communication reliability cannot always be guaranteed due to limitations of edge devices, network conditions, or transmission protocols. As a result, temporary measurement losses may occur even when the ingestion infrastructure is correctly designed.

For this reason, relying exclusively on communication reliability is not sufficient to guarantee continuous data streams. In monitoring applications where uninterrupted data availability is required, mechanisms capable of compensating for missing measurements become necessary.

This observation motivates the introduction of the predictive module described in the following section, which is designed to generate surrogate values when measurements are temporarily unavailable.

Nevertheless, in order to accurately evaluate the behavior of the subscriber architecture, it is necessary to isolate the subscriber from the publication bottleneck. For this reason, the following experiments enforce reliable message delivery by using QoS 1 and publisher-side acknowledgment.

Instead of increasing the message rate of a single sensor, the workload is increased by deploying multiple sensors transmitting at moderate frequencies. This approach allows the subscriber to be stressed while avoiding artificial message loss caused by publisher-side limitations.

5.2.8 Subscriber Scalability Under Controlled Workload

The previous experiments showed that increasing the message generation rate may introduce message loss caused by limitations in the publisher itself. In order to accurately evaluate the behavior of the subscriber architecture, it is therefore necessary to isolate the subscriber from publication-side bottlenecks.

To achieve this, a controlled experimental setup was introduced. In this configuration, message delivery reliability is enforced using QoS 1, ensuring that each message is acknowledged by the broker before the publisher proceeds with the next transmission. This mechanism guarantees that all generated measurements reach the MQTT broker, eliminating publisher-side losses.

Instead of increasing the transmission rate of a single sensor, the workload is increased by deploying multiple sensors transmitting at a stable frequency of four measurements per second. In this way, the aggregate input rate is controlled by adjusting the number of sensors while maintaining a stable and reliable publication process.

This setup allows the ingestion pipeline to be stressed while ensuring that any observed degradation originates from the subscriber architecture itself.

Figure 5.15 reports the message loss rate observed as the aggregate input rate increases for both the synchronous subscriber and the concurrent subscriber based on a thread pool.

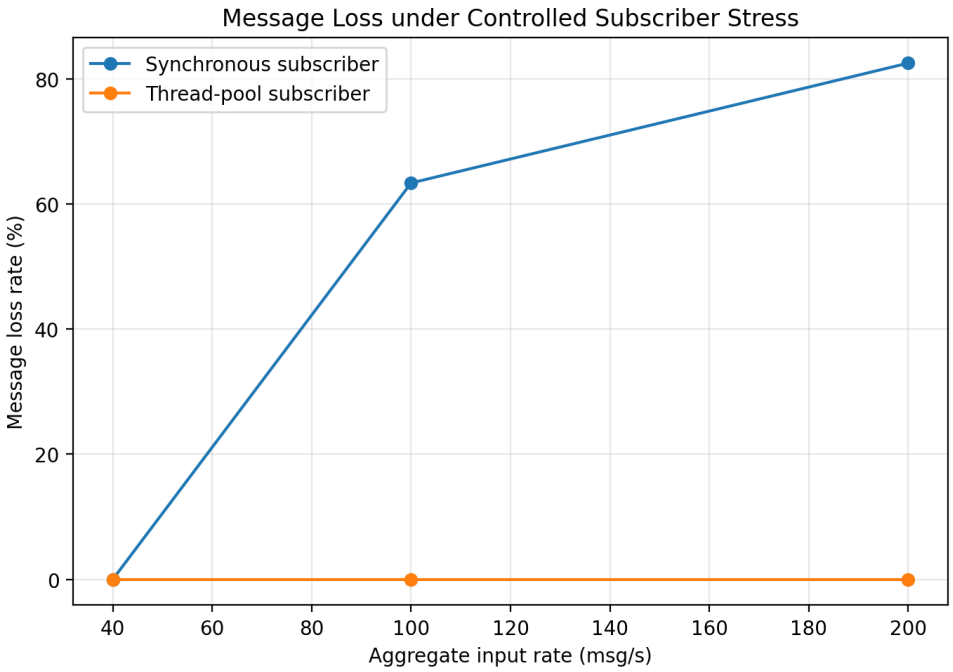

Figure 5.15: Message loss rate as a function of the aggregate input rate for synchronous and thread-pool subscriber architectures.

The results clearly highlight the limitations of the synchronous subscriber architecture. When the aggregate input rate increases beyond approximately 25 messages per second, the subscriber is no longer able to process incoming messages fast enough. As a consequence, a large fraction of the measurements is dropped.

This behavior is consistent with the processing cost of the persistence operation, which requires approximately 40 ms per message. Under sequential processing, this limits the sustainable throughput of the subscriber to roughly 25 messages per second.

In contrast, the concurrent subscriber architecture significantly improves the processing capacity of the system. By delegating message processing and persistence operations to multiple worker threads, the subscriber can process several messages simultaneously, avoiding the sequential bottleneck.

Figure 5.16 shows the effective throughput achieved by the two architectures under increasing input rates.

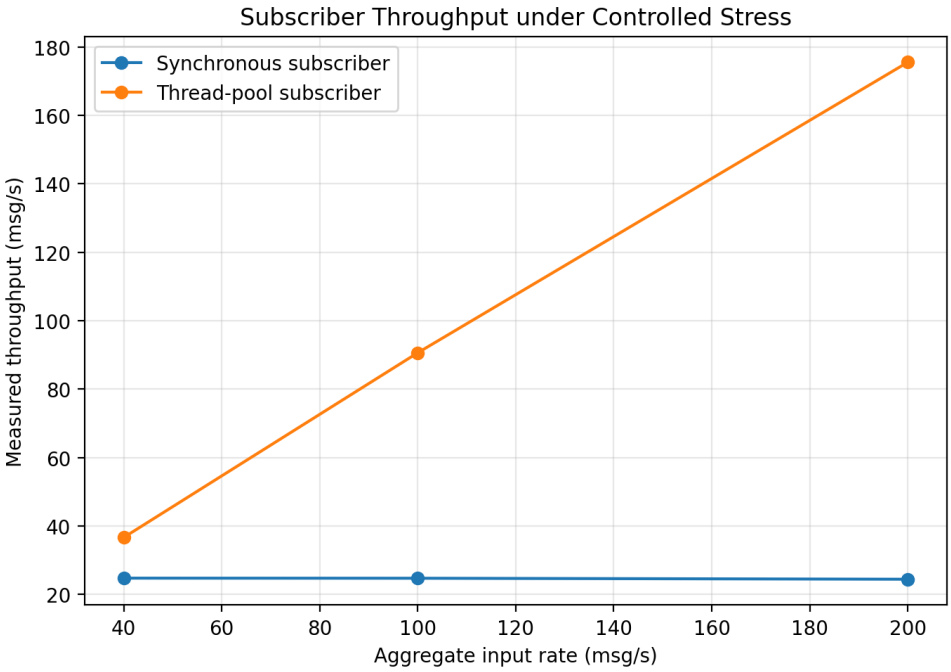

Figure 5.16: Measured subscriber throughput under increasing aggregate input rates.

While the synchronous subscriber quickly saturates at approximately 25 messages per second, the thread-pool subscriber is able to scale with the incoming workload and sustain significantly higher throughput. Even when the aggregate input rate reaches 200 messages per second, the concurrent architecture continues to process the incoming stream without introducing message loss.

The effect of queue accumulation is also visible in the delivery latency observed for the synchronous architecture. As shown in Figure 5.17, the delivery latency rapidly increases when the subscriber enters a saturated regime.

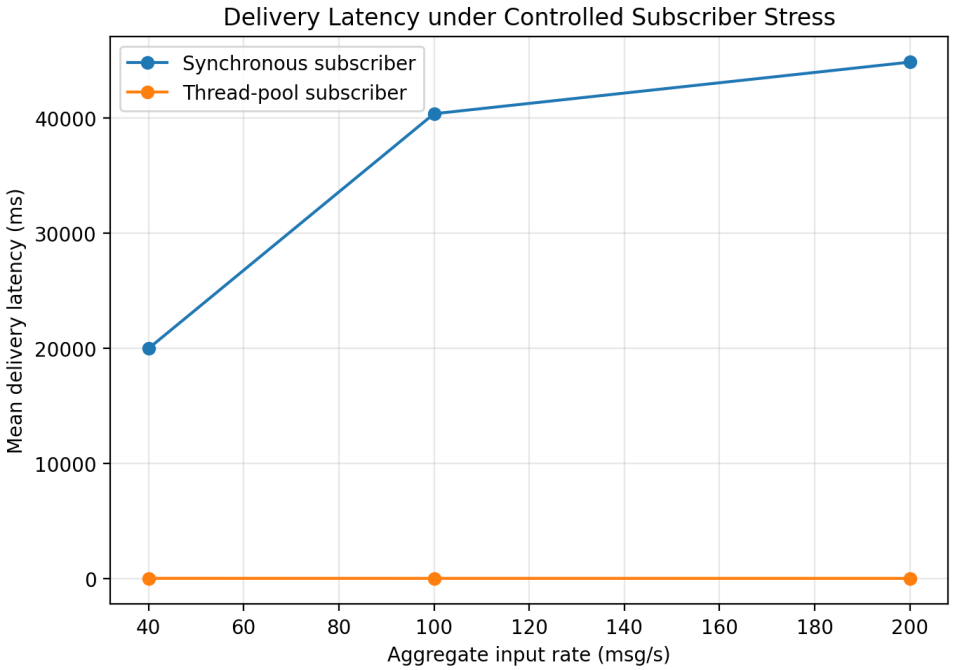

Figure 5.17: Delivery latency under increasing aggregate input rates for synchronous and concurrent subscriber architectures.

These results demonstrate that the main limitation of the synchronous subscriber architecture is the sequential processing model. By introducing concurrent workers, the thread-pool subscriber effectively removes this bottleneck and significantly increases the ingestion capacity of the pipeline.

5.2.9 Discussion and System Capacity

The experiments presented in this section highlight how both the communication-layer design and the subscriber architecture influence the overall scalability of the ingestion pipeline.

From the perspective of message routing, the progressive refinement of the topic organization significantly reduces the workload delivered to individual subscribers. Initially, in the baseline configuration, all measurements are delivered through a single topic, forcing the subscriber to receive the complete stream generated by the sensing infrastructure. As the topic structure is refined, the amount of traffic handled by each subscriber is progressively reduced.

Table 5.1 summarizes this evolution of the ingestion workload.

Configuration	Messages received per subscriber	Source of reduction
Single topic (baseline)	$N_{sensors} \times N_{features}$	All measurements share one topic
Metric filtering	$N_{sensors} \times 1$	Subscriber discards irrelevant metrics
Metric topics	$N_{sensors}$	Broker-side filtering by metric
Hierarchical topics	≈ 1 stream per subscriber	Partitioning by location and metric

Table 5.1: Evolution of the ingestion workload as the topic organization is progressively refined.

By partitioning the data stream according to both metric and location, the hierarchical topic structure effectively decouples the workload of each subscriber from the total number of deployed sensors. This design enables horizontal scalability by allowing the system to distribute different portions of the sensing infrastructure across multiple independent consumers.

In addition to routing design, the architecture of the subscriber itself also plays a critical role in determining the processing capacity of the ingestion pipeline. Under sequential processing, each message must be fully processed before the next one can be handled. In the experimental setup, the persistence operation requires approximately 40 ms per message, which limits the sustainable throughput of the synchronous subscriber to roughly 25 messages per second.

To overcome this limitation, the subscriber architecture was extended with a thread pool that allows multiple persistence operations to be executed concurrently. Table 5.2 summarizes the processing capacity observed for the two architectures under controlled workload conditions.

Architecture	Workers	Max sustainable rate	Observed loss
Synchronous subscriber	1	≈ 25 msg/s	High above threshold
Thread pool subscriber	4	≈ 175 msg/s	None observed up to 200 msg/s

Table 5.2: Subscriber processing capacity under controlled workload conditions.

The results clearly show that the synchronous subscriber quickly reaches saturation when the incoming message rate exceeds its sequential processing capacity. In contrast, the concurrent subscriber architecture significantly increases the ingestion capacity of the pipeline by allowing multiple messages to be processed simultaneously.

However, the experiments also highlight an important limitation of communication-layer optimizations. As shown in the previous experiments, message loss may still occur due to limitations in the publisher or in the transmission process itself. In particular, under high-frequency conditions the publisher may fail to sustain the requested transmission rate, causing measurements to be dropped before reaching the ingestion pipeline.

This observation reflects a common characteristic of IoT monitoring systems. In real-world deployments, perfect communication reliability cannot always be guaranteed due to constraints imposed by edge devices, network conditions, or communication protocols. As a consequence, temporary measurement losses may occur even when the ingestion architecture is correctly designed and properly scaled.

For applications that require continuous monitoring and data availability, relying exclusively on communication reliability is therefore insufficient. Instead, additional mechanisms are required to preserve the continuity of the data stream when measurements are temporarily unavailable.

For this reason, the proposed framework complements the ingestion pipeline with a predictive module capable of generating surrogate values when measurements are missing. The following section therefore evaluates the forecasting models used to maintain data continuity in the presence of temporary data loss.

5.3 Forecast-Based Data Continuity

The previous section analyzed the scalability and performance limits of the data ingestion pipeline under increasing workload conditions. The results showed that architectural improvements such as topic reorganization and concurrent processing significantly increase the throughput of the system. However, the experiments also highlighted a fundamental limitation that cannot be entirely eliminated at the communication layer. When the message generation rate becomes sufficiently high, some measurements may still be lost before reaching the ingestion pipeline. In particular, packet loss can occur directly at the publisher side under high-frequency transmission conditions. Publishers are then unable to sustain the transmission rate required to maintain a perfectly continuous stream of measurements.

A straightforward solution would be to throttle the publishers or to introduce back-pressure mechanisms that slow down the data generation rate when congestion occurs. While this approach could reduce packet loss, it would also directly impact the temporal resolution of the collected data. In many monitoring scenarios, however, maintaining a stable and predictable sampling frequency is a key requirement.

For this reason, instead of modifying the sensing layer, the proposed framework introduces a strategy aimed at preserving the continuity of the data stream even when measurements are temporarily missing. The idea is to exploit short-term forecasting models to generate surrogate values whenever real measurements are unavailable. When a measurement is missing due to packet loss or temporary communication disruptions, the predictive module estimates the expected value based on the recent history of the time series. These predicted values are then inserted into the stream so that downstream services can operate without interruption.

The remainder of this section evaluates the forecasting models used to implement this mechanism. Both prediction accuracy and runtime cost are analyzed.

5.3.1 Experimental Setup

The evaluation focuses on temperature measurements collected within the monitored environment introduced in Chapter 3. The objective is to assess the ability of different models to provide accurate short-term predictions that can replace missing measurements during temporary data gaps.

In the real system, missing measurements may arise due to packet loss, temporary network disruptions, or saturation of the data publishers. To emulate such conditions in a controlled experimental setting, missing observations were artificially introduced into an otherwise complete dataset.

Two injection strategies were considered. The first strategy simulates systematic congestion patterns by periodically removing observations from the sequence (e.g., every Nth measurement). The second strategy introduces stochastic packet loss by randomly removing a fixed percentage of observations from the time series.

The predictive module operates continuously alongside the ingestion pipeline. As new observations arrive, the forecasting model updates its internal state using the most recent measurements. When a gap is detected in the incoming data stream, the predictive module does not need to be activated explicitly; instead, the most recent forecast produced by the model is used as a surrogate value and inserted into the stream. In this way, downstream components continue to receive a temporally consistent sequence of measurements.

The forecasting experiments use the same environmental dataset adopted for the pipeline evaluation, resampled to a uniform sampling interval of one minute. This choice reflects the relatively slow dynamics of temperature measurements and avoids unnecessary computational overhead.

With a one-minute sampling interval, the forecasting horizons $H \in \{1, 10, 30, 60\}$ correspond to temporal gaps of 1, 10, 30, and 60 minutes, respectively. The goal of the predictive module is not long-term forecasting but short-term gap filling: the model is expected to reconstruct plausible values when measurements are temporarily unavailable.

The choice of multiple forecasting horizons reflects the uncertainty regarding the duration of possible data interruptions in real IoT deployments. Packet loss may lead to isolated missing measurements, but longer gaps may also occur if connectivity is degraded for a short period of time. Since the duration of such interruptions cannot be determined a priori, the evaluation considers several horizons in order to analyze how forecasting models behave under progressively longer gaps.

Several forecasting approaches were evaluated, including both classical statistical models and neural architectures. In particular, the experiments consider ARIMA models with

different parameterizations and LSTM-based neural networks with multiple architectural variants and history window sizes.

Each model is trained on the first 80% of the time series and evaluated on the remaining 20% while the artificial gaps are introduced during the evaluation phase.

For each model we report multiple evaluation metrics. Prediction accuracy is quantified using the Mean Absolute Error (MAE) and the Root Mean Squared Error (RMSE). Since both metrics lead to consistent model rankings, the discussion primarily focuses on MAE values.

In addition to accuracy, the evaluation also considers the runtime behavior of the models. In a streaming system, forecasting models must continuously update their internal state as new observations arrive and must be able to produce predictions with minimal latency when measurements are missing.

For this reason, two latency components are measured:

- **Prediction latency**, representing the time required to generate a forecast given the current model state;

- **Update latency**, representing the time required to update the model state when a new observation becomes available.

These two metrics capture the operational cost of integrating forecasting models within a real-time IoT data pipeline.

5.3.2 Prediction Accuracy

The first aspect considered in the evaluation concerns the ability of the forecasting models to reconstruct missing measurements with sufficient accuracy. Since the predictive module replaces temporarily unavailable observations, the quality of the generated values directly determines the plausibility of the reconstructed data stream.

Figure 5.18 summarizes the prediction accuracy obtained by the evaluated models across the considered forecast horizons. Because the sampling interval is one minute, the evaluated horizons correspond to gaps of 1, 10, 30, and 60 minutes.

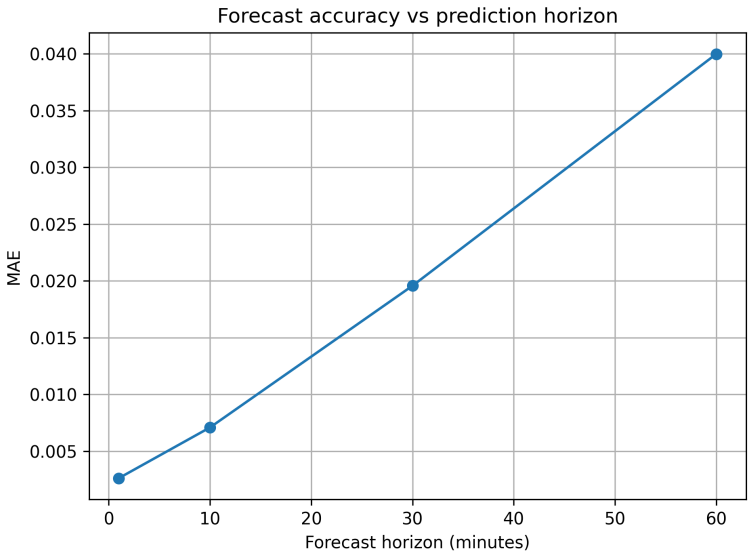

Figure 5.18: Prediction accuracy across different forecasting horizons.

Overall, classical statistical models remain highly competitive for short-term forecasting tasks. The ARIMA(5,1,1) model with FFT pre-processing (denoted *ARIMA_511_FFT*) consistently achieves the lowest prediction error across all evaluated horizons.

Table 5.3 summarizes the best model for each horizon together with the corresponding MAE and latency values. For a horizon of $H = 1$, the ARIMA model attains a MAE of 2.6×10^{-3}, whereas the best LSTM model (a medium-sized dense LSTM with a 240-sample history window) achieves a MAE of 3.1×10^{-3}. The difference corresponds to an 18.7% relative improvement in favor of ARIMA.

As the horizon increases to $H = 10$, the ARIMA model maintains a MAE of 7.1×10^{-3}, compared with 7.6×10^{-3} for the best LSTM model, yielding a 7.6% improvement.

For $H = 30$ and $H = 60$, ARIMA continues to outperform the neural models by approximately 15% and 24%, respectively, demonstrating that the gap in accuracy widens with the forecast horizon.

An additional observation concerns the influence of the input window size used by the neural networks. LSTM models trained with longer history windows generally produce lower MAE values. For example, at $H = 1$ the average MAE across LSTM variants with a 240-sample window is about 1.2×10^{-2}, whereas reducing the window to 60 samples increases the error to 1.3×10^{-2}. Similar trends hold for longer horizons, indicating that a longer temporal context improves the ability of neural models to capture the dynamics of the monitored environment. However, even the best LSTM configuration does not surpass the ARIMA model for the series considered in this study.

These results indicate that statistical models such as ARIMA provide consistently strong performance for short-term gap filling in slowly varying environmental signals.

Table 5.3: Best forecasting model for each prediction horizon.

Horizon	Model	MAE	RMSE
1	ARIMA(5,1,1) + FFT	0.0026	0.0026
10	ARIMA(5,1,1) + FFT	0.0071	0.0082
30	ARIMA(5,1,1) + FFT	0.0196	0.0233
60	ARIMA(5,1,1) + FFT	0.0400	0.0482

Nevertheless, prediction accuracy alone is not sufficient to determine the suitability of a forecasting model within a real-time IoT pipeline. The computational cost of operating the model continuously must also be considered.

5.3.3 Runtime Performance

In addition to prediction accuracy, it is important to evaluate the runtime cost associated with each forecasting model. Since the models are intended to operate within a real-time streaming pipeline, both prediction latency and update latency must be considered. The latency breakdown in Figure 5.19 reveals a clear difference between statistical and neural approaches.

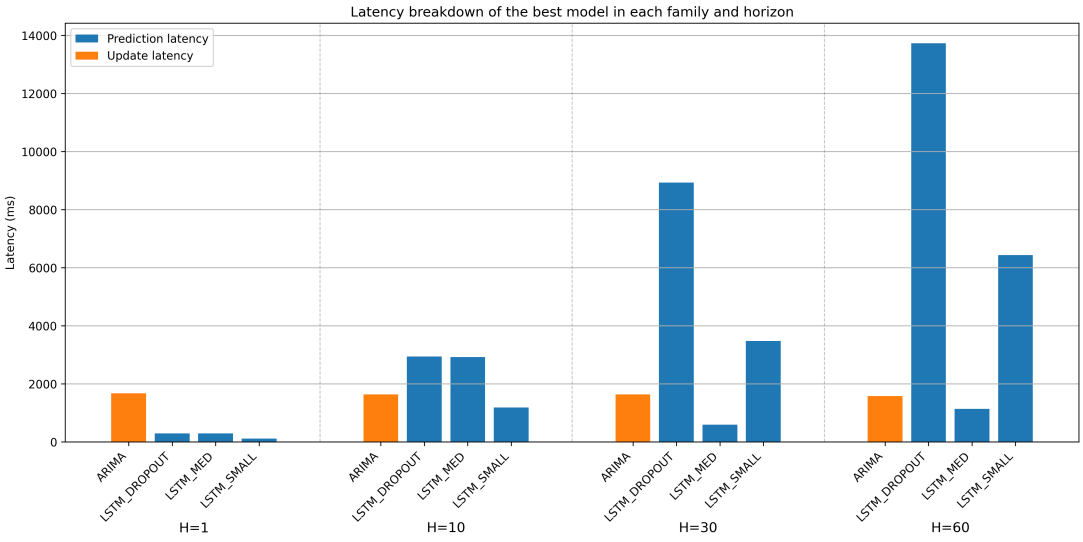

Figure 5.19: Prediction and update latency for the best model of each family across the evaluated forecast horizons.

Table 5.4: Average prediction and update latency for the best models.

Model	Horizon	Prediction latency (ms)	Update latency (ms)
ARIMA	1	2.5	1670
ARIMA	10	3.1	1620
ARIMA	30	4.8	1630
ARIMA	60	8.9	1560
LSTM	1	294	1.2
LSTM	10	1200	1.3
LSTM	30	3400	1.3
LSTM	60	13700	1.4

ARIMA models exhibit extremely low prediction latency because the forecast for a given horizon can be computed in closed form without iterating over each predicted step; prediction times remain below 9 ms even for a 60-step horizon. However, updating the model after each new observation requires re-estimating the ARIMA parameters. This re-estimation is computationally intensive and dominates the runtime cost: the mean update latency is on the order of 1.6 s, more than two orders of magnitude larger than the prediction time.

Neural models based on LSTM architectures exhibit a complementary behavior. For these models the update step amounts to shifting the sliding input window and does not involve parameter re-training, which results in an update latency around 1 ms. The prediction phase, however, requires sequentially propagating the input through several LSTM layers and computing one output per forecast step. Because of the inherent architectural complexity of LSTMs—multiple gates and memory cells that must be evaluated at every time step—the mean prediction latency grows linearly with the forecast horizon. In our experiments, prediction latency increases from approximately 294 ms for a single-step forecast to about 13.7 s for a 60-step horizon. These results are consistent with the broader literature, which notes that LSTMs offer flexible modeling capabilities but at the cost of higher computational requirements and slower inference speed.

5.3.4 Accuracy–Latency Trade-off

The relationship between forecasting accuracy and computational cost is visualized in Figure 5.20.

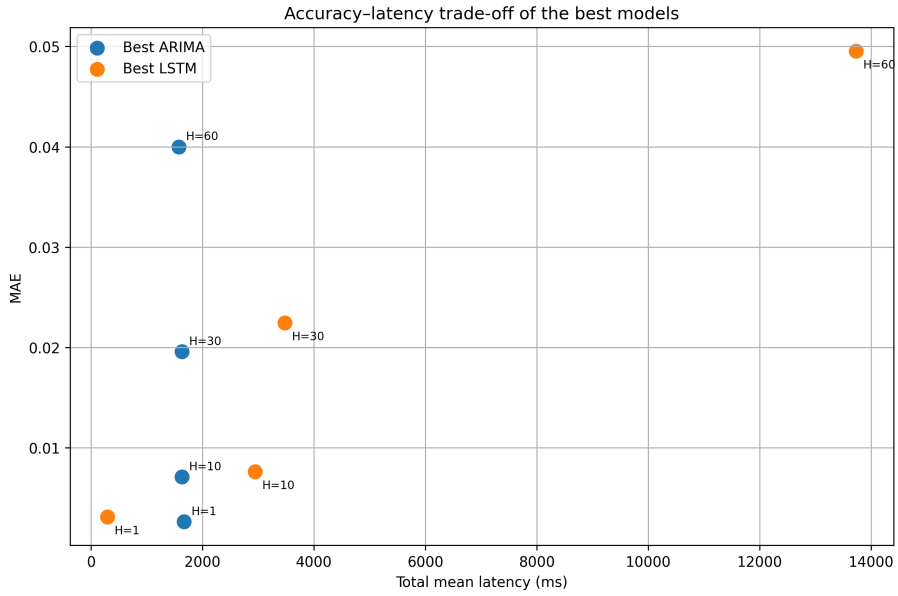

Figure 5.20: Accuracy–latency trade-off for the best ARIMA and LSTM models across the evaluated forecast horizons.

Models located closer to the origin of the plot achieve a more favorable balance between prediction accuracy and runtime latency. The ARIMA models occupy the region with the lowest prediction error and very low inference latency. Nevertheless, their high update latency places them farther from the origin along the vertical axis when frequent model updates are required. Conversely, the LSTM models exhibit slightly higher prediction error but extremely low update latency. Their total runtime cost is dominated by the prediction phase and grows with the forecast horizon; as a result, for horizons beyond $H = 30$ the total latency of LSTM models exceeds several seconds. The empirical trade-off illustrates that no single model simultaneously maximizes accuracy and minimizes all latency components.

5.3.5 Implications for Batch Prediction

An additional observation emerging from the latency analysis concerns the operational strategy adopted when generating forecasts in a streaming environment. The total computational cost of forecasting can be decomposed into two components:

$$C_{total} = C_{update} + C_{prediction}.$$

As discussed in the previous section, statistical models such as ARIMA exhibit extremely fast prediction times but relatively expensive model updates, since parameter re-estimation must be performed when new observations are incorporated. Conversely, LSTM-based models require negligible update time but significantly higher prediction latency, which increases with the forecasting horizon.

These characteristics suggest different operational strategies for the two model families. In particular, when update operations are computationally expensive, it may be advantageous to reduce the frequency of updates and instead generate forecasts in batches. In this configuration, the model is updated periodically using the most recent observations, after which a multi-step forecast is computed to cover the expected duration of potential data gaps.

For models such as ARIMA, this strategy can significantly reduce the amortized cost per prediction, since the expensive update operation is performed only once for multiple forecasted steps. By contrast, LSTM models do not benefit as strongly from this strategy, as their update cost is already negligible while prediction latency grows with the forecast horizon.

These observations further motivate the runtime-aware model selection approach adopted in the predictive module. Depending on the frequency of incoming measurements and the expected duration of data gaps, the system may favor models that minimize update cost or models that minimize prediction latency.

5.3.6 Discussion

The experimental results highlight that the choice of a forecasting model cannot be based solely on prediction accuracy. In real-time data processing systems, runtime constraints such as inference latency and model update cost play an equally important role. ARIMA models provide excellent prediction accuracy and very fast inference, confirming their effectiveness for short-term forecasting of stationary series [19]. However, their update latency is more than two orders of magnitude larger than the prediction latency, because re-estimating the model parameters with each new observation is computationally intensive [20]. These high update costs make ARIMA less suitable in scenarios where the model must be updated after every measurement.

LSTM models, although slightly less accurate, offer a different performance profile. Their update latency is negligible because they operate in a stateful manner—simply appending the new observation to the input window—while their prediction latency scales linearly with the forecasting horizon. The higher computational demand and slower inference speed of LSTMs mean that their overall runtime can exceed that of ARIMA models for long prediction horizons, despite the minimal update cost. Nevertheless, in scenarios where predictions are required at a very high frequency and model updates occur continuously, the low update latency of LSTM models may compensate for their increased prediction time.

These observations motivate the runtime-aware model selection strategy adopted in the predictive module. Rather than relying on a single forecasting model, the framework can dynamically select the approach that best satisfies the operational constraints of

the system while maintaining an adequate level of prediction accuracy. For example, in applications where the model is updated only occasionally (e.g., when environmental conditions change slowly), the ARIMA model may be the preferred choice because of its superior accuracy and low inference latency. Conversely, in settings where the model state must be updated continuously (e.g., high-frequency sensor streams), an LSTM variant with an appropriate history window can offer a more balanced trade-off between accuracy and runtime cost. This adaptive behaviour allows the predictive module to maintain continuity of the data stream by inserting surrogate values when gaps are detected, without incurring unnecessary computational overhead.

Chapter 6

Conclusions and Future Work

This thesis addressed the problem of maintaining data continuity in real-time IoT systems under temporary disconnections. While existing solutions improve delivery reliability through buffering and retransmission, or reconstruct missing data offline through imputation techniques, they do not explicitly ensure that data streams remain continuous during outages. However, many real-time applications require a stable and regular flow of data at a fixed sampling rate, making **stream continuity a critical requirement**.

To address this gap, this work proposed a **predictive fallback approach**, where forecasting models are used to generate temporary substitute values when real measurements are unavailable. Unlike traditional imputation methods, the proposed approach explicitly incorporates runtime constraints, requiring predictive models to operate within the temporal limits imposed by the sampling interval.

6.1 Summary of Contributions

The main contributions of this thesis can be summarized as follows:

- **Design of a real-time IoT data pipeline.** A scalable ingestion architecture based on MQTT was developed, supporting hierarchical topic organization, message filtering, and concurrent processing through a thread pool. This design enables efficient handling of high-frequency sensor streams while reducing bottlenecks in the subscriber component.

- **Integration of a predictive fallback mechanism.** A predictive module was introduced to generate short-term forecasts during temporary disconnections, enabling the system to preserve stream continuity rather than relying solely on delayed data delivery.

- **Runtime-aware evaluation of forecasting models.** The work systematically evaluates both statistical (ARIMA) and neural (LSTM) models, highlighting the

importance of considering inference latency and update cost in addition to prediction accuracy.

- **Formulation of a feasibility constraint.** A practical condition linking model inference latency to the sampling interval was introduced, providing a clear criterion for determining whether a predictive model can be deployed in real-time IoT scenarios.

6.2 Key Results

Experimental results highlight several important findings:

- **Pipeline scalability.** A synchronous subscriber becomes a bottleneck as the number of sensors increases, leading to rapid growth in latency and message loss. Introducing parallel processing significantly improves throughput and delays saturation, enabling the system to sustain higher workloads.
- **Continuous service operation.** The integration of predictive fallback allows the system to maintain a continuous data stream even during disconnections, avoiding gaps that would otherwise disrupt downstream applications.
- **Accuracy–latency trade-off.** ARIMA models achieve higher prediction accuracy and extremely low inference latency, making them well suited for short-term forecasting under strict timing constraints. However, their high update cost limits their applicability in rapidly evolving scenarios. Conversely, LSTM models provide a more balanced trade-off, with slightly lower accuracy but faster updates and greater flexibility.

These results demonstrate that model selection in real-time IoT systems cannot rely solely on prediction accuracy, but must account for the interaction between model behavior and system-level constraints.

6.3 Limitations

Despite the promising results, several limitations remain:

- **Static disconnection detection.** The use of fixed thresholds for detecting disconnections may lead to suboptimal behavior under variable network conditions.
- **Lack of online learning.** Predictive models are trained offline and do not adapt to changes in data distribution over time.
- **Limited evaluation scope.** The experimental validation focuses on temperature data from a single environment, limiting the generality of the results.

- **Absence of explicit reconciliation mechanisms.** While the system is designed to support the replacement of predicted values with real measurements, no explicit reconciliation component has been implemented. This limits the ability to ensure full consistency between predicted and actual data after disconnections.
- **Centralized infrastructure.** The current implementation relies on centralized components such as the MQTT broker and ingestion services, which may represent single points of failure.

6.4 Future Work

Several directions can be explored to extend the proposed framework.

Extension to additional predictive models. The modular design of the predictive component allows new forecasting models to be integrated with minimal effort through the defined interface. Future work could evaluate a broader range of models, including lightweight online algorithms or hybrid approaches, to further explore the trade-off between prediction accuracy and runtime constraints. This would also enable more advanced runtime selection strategies based on a richer model pool.

Persistence optimization and storage-aware design. The experimental evaluation highlighted that data persistence may become a bottleneck in high-frequency scenarios, due to the non-negligible cost of write operations. Although the current system mitigates this through pipeline-level optimizations, a deeper integration with storage systems could improve performance. Future work could investigate batching strategies, asynchronous persistence mechanisms, or time-series databases optimized for high-throughput ingestion, in order to reduce latency and prevent backpressure effects.

Adaptive disconnection detection. The current system relies on fixed thresholds to detect data interruptions. More advanced approaches based on statistical or learning-based methods could dynamically adapt detection thresholds to varying network conditions, improving robustness and reducing false positives.

Online learning and model adaptation. In long-running deployments, changes in environmental conditions or system behavior may degrade prediction accuracy. Incorporating online learning or periodic retraining mechanisms would allow predictive models to adapt to concept drift and maintain performance over time.

Integration of reconciliation mechanisms. Future work should introduce explicit reconciliation strategies to replace predicted values with real measurements once they become available. Given the modular and decoupled nature of the publish/subscribe architecture, such mechanisms can be implemented as independent components operating on the data stream.

Additional Experimental Details

.1 Extended Forecasting Results

This section reports additional forecasting results obtained during the experimental evaluation. While Chapter 5.3 focuses on the most relevant configurations, the following tables provide a more detailed overview of the evaluated models.

Table 1 summarises a representative subset of the evaluated configurations, including both ARIMA and LSTM variants across different forecasting horizons. Due to space constraints, only a subset of the evaluated configurations is reported here, while the complete set of results is available in the accompanying experimental data files.

Table 1: Representative subset of forecasting results across model configurations.

Model	**Family**	**H**	**MAE**	**Pred Lat. (ms)**	**Update Lat. (ms)**
ARIMA_511_FFT	ARIMA	1	0.0026	6.74	1672.57
ARIMA_311	ARIMA	1	0.0028	3.45	1317.74
LSTM_DENSE_MED_W240_H1	LSTM_MED	1	0.0031	293.53	1.11
LSTM_DROPOUT_W240_H1	LSTM_DROPOUT	1	0.0032	292.07	1.14
LSTM_DENSE_MED_W60_H1	LSTM_MED	1	0.0032	164.40	1.39

Table 2 presents the latency characteristics of different model families, highlighting the trade-off between computational cost and predictive accuracy.

Table 2: Latency summary by model family and forecasting horizon.

H	Family	Pred Lat. (ms)	Update Lat. (ms)
1	ARIMA	6.74	1672.57
1	LSTM_MED	293.53	1.11
1	LSTM_DROPOUT	292.07	1.14
1	LSTM_SMALL	116.02	1.35
10	ARIMA	4.77	1629.34
10	LSTM_MED	2921.77	1.14
10	LSTM_DROPOUT	2942.18	1.25
10	LSTM_SMALL	1183.86	1.59
30	ARIMA	0.00	1631.48
30	LSTM_MED	591.23	1.38
30	LSTM_DROPOUT	8931.94	1.23
30	LSTM_SMALL	3474.01	1.62
60	ARIMA	1.02	1575.82
60	LSTM_MED	1142.52	1.33
60	LSTM_DROPOUT	13732.73	1.17
60	LSTM_SMALL	6430.80	1.64

The results confirm that ARIMA models exhibit significantly higher update latency compared to LSTM-based approaches, while maintaining competitive prediction accuracy, as discussed in Chapter 5.

.2 Forecasting Model Configuration

The forecasting models evaluated in this work include both statistical and neural approaches, configured to operate under real-time constraints.

ARIMA Models Two ARIMA configurations were considered:

- ARIMA(3,1,1)
- ARIMA(5,1,2) with Fourier terms (period = 1440, K = 3) to capture daily seasonality

LSTM Models Multiple LSTM variants were evaluated, differing in architecture and input configuration:

- Hidden units: 64 and 128
- Dense layers: 64 and 128 units
- Dropout variant with rate 0.2

The models were trained using the following parameters:

- Optimizer: Adam (learning rate = 0.001)
- Loss function: Huber loss
- Batch size: 32
- Maximum epochs: 200
- Early stopping with patience = 10
- Learning rate reduction on plateau

The evaluation considered multiple input window sizes and forecasting horizons:

- Window sizes: 60, 120, 240
- Forecast horizons: 1, 10, 60

All models were trained and evaluated using a rolling forecasting origin protocol, as described in Chapter 4.

.3 Experimental Setup

All experiments were conducted on a computing environment equipped with [INSERT CPU], [INSERT RAM], and running [INSERT OS].

The forecasting experiments were performed using Python, leveraging TensorFlow/Keras for neural models and standard statistical libraries for ARIMA implementations.

The dataset consists of temperature measurements collected from a real IoT deployment, sampled at a frequency of one observation per minute.

The evaluation follows a temporal split, with 80% of the data used for training and the remaining 20% for testing. Models are evaluated using a rolling forecasting origin protocol to simulate real-time prediction scenarios.

Latency measurements were obtained by instrumenting both prediction and update operations during runtime, capturing the computational cost of each model under realistic streaming conditions.

Bibliography

[1] 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.

[2] S. Kahveci, B. Alkan, M. H. Ahmad, B. Ahmad, and R. Harrison, "An end-to-end big data analytics platform for IoT-enabled smart factories," *Journal of Manufacturing Systems*, vol. 64, pp. 214–226, 2022.

[3] OASIS Standard, "MQTT Version 3.1.1," 2014. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[4] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital Twin in manufacturing: A categorical literature review," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[5] M. J. Azur, E. A. Stuart, C. Frangakis, and P. J. Leaf, "A Tutorial on Multilevel Multiple Imputation of Missing Data," *International Journal of Methods in Psychiatric Research*, vol. 20, no. 1, pp. 40–49, 2011.

[6] R. J. A. Little and D. B. Rubin, *Statistical Analysis with Missing Data*. Wiley, 2002.

[7] OASIS Standard, "MQTT Version 5.0," 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[8] Microsoft, "Azure IoT Edge Documentation," 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/iot-edge/

[9] Amazon Web Services, "AWS IoT Greengrass Stream Manager," 2024. [Online]. Available: https://docs.aws.amazon.com/greengrass/

[10] OPC Foundation, "OPC UA Specification," 2024. [Online]. Available: https://reference.opcfoundation.org/

[11] W. Cao, D. Wang, J. Li, H. Zhou, L. Li, and Y. Li, "BRITS: Bidirectional Recurrent Imputation for Time Series," *NeurIPS*, 2018.

[12] A. Cini, I. Marisca, and C. Alippi, "GRIN: Graph Neural Networks for Multivariate Time Series Imputation," *arXiv preprint*, 2021.

[13] Apache Software Foundation, "Apache Flink Dynamic Tables," 2024. [Online]. Available: https://flink.apache.org

[14] M. Fragkoulis *et al.*, "A Survey on Stream Processing Systems," *ACM Computing Surveys*, 2020.

[15] Anonymous, "Missing Data Imputation for IoT Edge Devices," *IEEE Access*, 2022.

[16] ——, "Online Missing Data Imputation in Sensor Streams," *PVLDB*, 2024.

[17] ——, "Imputation Techniques for IoT Gateways," *Sensors*, 2021.

[18] Apache Kafka, "Kafka Log Compaction," 2024. [Online]. Available: https://kafka.apache.org/documentation/

[19] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Wiley, 2015.

[20] R. J. Hyndman and G. Athanasopoulos, *Forecasting: Principles and Practice*. OTexts, 2018. [Online]. Available: https://otexts.com/fpp3/

[21] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[22] S. B. Taieb and G. Bontempi, "A Review and Comparison of Strategies for Multi-step Ahead Time Series Forecasting," *arXiv preprint*, 2012.