POLITECNICO DI TORINO

**MASTER's Degree in COMPUTER ENGINEERING**


# Architectural Refactoring: from SQS Message Processing to Typed Event-Bus Registry

**Supervisors**

Prof. ANTONIO VETRO'

**Candidate**

MARIO VARAVALLO

**JULY 2026**

# Abstract

Most modern distributed systems increasingly rely on asynchronous, event-driven communication to achieve scalability and decouple services. But, when messages are exchanged as untyped structures and processed through centralized dispatching logic, systems often become difficult to maintain, extend, and also validate, leading to runtime fragility and reduced reliability.

This thesis presents the design and implementation of a Typed Event-Bus Registry architecture built on top of an existing Amazon SQS-based infrastructure. The proposed solution introduces strongly typed event definitions using Pydantic models, which enable explicit schema validation and improve type safety across the asynchronous communication layer. In addition, a registry-based dispatching mechanism is introduced to dynamically resolve and invoke event handlers, replacing a traditional centralized dispatcher based on if/else conditional logic.

The resulting architecture improves separation of concerns, validation, routing, and execution. This leads to increased maintainability, extensibility, and reliability of the system, while preserving compatibility with the existing messaging infrastructure.

The refactored architecture shows that the proposed approach effectively addresses the limitations of the original implementation, particularly in terms of weak schema enforcement and tight coupling in event handling logic. Although the new architecture introduces additional abstraction layers and development overhead, these problems are justified by the long-term benefits in system robustness and clarity.

Overall, the work demonstrates how structured event modeling and a registry-based dispatching strategy can significantly enhance the design of event-driven distributed systems based on microservices.

# Contents

# Chapter 1

# Introduction

## 1.1 Context and Motivation

Modern software systems are increasingly required to handle large-scale workloads and ensure high availability and rapid evolution. Traditional monolithic architectures, while effective in early development stages and within small domains, often struggle to meet these requirements when the system's complexity grows.

The rise of distributed systems and microservices architectures has enabled organizations to build scalable and resilient applications by decomposing functionality into smaller, independent services, each divided into a meaningful domain. These services communicate over networks and can be developed, deployed, and scaled independently from each other.

In this broad context, asynchronous communication mechanisms such as message queue strategies play a fundamental role. In particular, systems based on Amazon SQS are widely adopted due to their reliability and scalability. However, such systems may also introduce other challenges related to data consistency, message validation, and system maintainability.

This thesis is motivated by the need to improve robustness and maintainability in distributed, event-driven systems that heavily rely on message-based communication.

## 1.2 Objectives of the Thesis

The main objective of this thesis is to design and implement a solution that improves the reliability and maintainability of event-driven distributed systems.

In particular, the goals of this work are:

- Introduce strong type safety in message-based communication by enforcing well-defined contracts for all exchanged messages. This reduces runtime errors, enables earlier detection of inconsistencies, and improves maintainability in distributed systems.
- Improve schema validation for exchanged events by ensuring that all messages conform to predefined structural and semantic rules. This includes validating required fields, data types, constraints, and handling schema versioning to maintain backward and forward compatibility.
- Reduce coupling between system components by decoupling services through asynchronous, message-based communication. This allows components to evolve, scale, and be deployed independently, improving system flexibility and resilience.
- Enhance observability and traceability of system behavior, enabling end-to-end visibility of message flows across services, facilitating debugging, monitoring, and auditing in complex distributed architectures.

To achieve all these objectives, a new Typed Event-Bus Registry architecture is proposed, making it possible to manage structured events and safer communication between different microservices.

## 1.3 Thesis Structure Overview

This thesis is organized as follows:

- **Chapter 2** presents the theoretical background of the work, including: distributed systems, microservices, event-driven architectures, and message queues.
- **Chapter 3** describes the existing system and highlights its current limitations, focusing on problematic issues related to type safety, schema enforcement, and system coupling.
- **Chapter 4** introduces the proposed solution, detailing the design of the Typed Event-Bus Registry and its architectural principles capable of improving the previous architecture.

- **Chapter 5** presents the actual implementation of the proposed system solution, including event modeling, the new registry design, and message processing.
- **Chapter 6** evaluates the solution, discussing its effectiveness and also limitations.
- **Chapter 7** provides a broader discussion about the new architecture, including comparisons with alternative approaches and possible future developments.
- **Chapter 8** concludes the thesis by summarizing its main results.

# Chapter 2

# Background and Theoretical Foundations

## 2.1 Distributed Systems Fundamentals

### 2.1.1 Monolithic vs Distributed Architectures

Software systems can be categorized into two categories: monolithic architectures and distributed architectures.

A monolithic architecture consists of a single, unified codebase where all components—such as user interface, business logic, and data access—are tightly coupled and deployed together. Adopting this approach simplifies the initial development and the deployment of the application, but becomes increasingly difficult to maintain and scale as the system grows and new features are added [1].

As the codebase expands and the system becomes larger and more complex, even small changes may require rebuilding and redeploying the entire application from scratch, increasing the risk of introducing unintended and dangerous side effects. This architecture pattern also makes it harder for multiple teams to work concurrently on different parts of the system, as dependencies between components can lead to conflicts and slower development cycles, increasing costs and human errors. Furthermore, scaling a monolithic application typically involves replicating the entire system, resulting in inefficient and bad resource utilization.

Over time, a monolith system can become difficult to understand and modify, especially for new hired developers that haven't familiarity with the preexisting codebase. The lack of clear boundaries between components may

also lead to poor code organization and increased technical debt. Despite showing these severe limitations, monolithic architectures can still be effective for small applications or for early development stages, where simplicity and rapid iteration are more important than scalability and flexibility.

In contrast, a distributed architecture is composed of multiple independent and autonomous services that communicate over some kind of network system. Each service is responsible for a specific functionality of the application and can be developed, deployed, and scaled independently from the others [2]. This improves the flexibility and scalability of the system, but introduces additional complexity in communication and coordination between different services. In this scenario, the coordination and the collaboration between the different service layers became very mandatory.

Monolithic systems can typically scale vertically by increasing the resources of a single machine, while distributed systems support horizontal scaling by replicating services across multiple nodes. Is important to note that distributed systems also offer better fault isolation and failure handling, as failures can be contained within individual services rather than affecting the entire system [3].

### 2.1.2 Characteristics of Distributed Systems

Distributed systems are characterized by a set of fundamental properties that distinguish them from traditional centralized monolith architectures. These characteristics introduce both advantages but also new challenges, requiring careful design and implementation choices [2].

- **Physical distribution:** Components of a distributed system are deployed across multiple physical or even virtual nodes and communicate through a network. Unlike monolithic systems, where components reside within the same environment, distributed components can reside in different environments and must rely on network-based communication to interact. This introduces different new problems like: latency, variability in response times, and the possibility of partial or complete communication failures between the components. As a result, systems must be explicitly designed to handle issues such as message loss, delays, and network partitions.

- **Concurrency:** Distributed systems inherently support concurrent execution, as multiple components operate simultaneously on different nodes, working in parallel. These components often interact asynchronously, without a global clock or strict execution order. While

concurrency improves system performance and responsiveness, it also introduces complexity in the coordination between the service layer. For these reasons, developers must address challenges such as race conditions, deadlocks, and data inconsistencies, typically using synchronization mechanisms, distributed locking, or eventual consistency models.

- **Autonomy and loose coupling:** Each component in a distributed system is designed to operate independently, with limited knowledge of other components. This loose coupling enhances modularity and allows individual services to be developed, deployed, and scaled without affecting the whole system. However, this independence requires: well-defined interfaces, standardized communication protocols, and robust API contracts to ensure correct interaction among services. Also, changes in one component must be carefully managed to avoid breaking compatibility with others.

- **Transparency:** A distributed system should appear to users as a single coherent system, hiding the complexity of its distributed nature. Different forms of transparency can be achieved, including location transparency (hiding where resources are located), replication transparency (hiding duplication of components), and failure transparency (masking failures when possible).

The properties discussed above highlight how distributed systems differ fundamentally from centralized monolithic architectures, emphasizing both their flexibility and the new complexity they introduce. While characteristics such as distribution, concurrency, and loose coupling enable modularity and adaptability, they also require careful handling of communication, coordination, and system behavior under possible failure conditions.

Among the various aspects involved in the design of distributed systems, some play a particularly critical role in determining their effectiveness and reliability. In particular, scalability, fault tolerance, and consistency represent key concerns that must be addressed. These characteristics ensure that a system can handle increasing workloads, remain operational in the presence of failures, and maintain a coherent view of data across multiple nodes.

For this reason, these aspects are analyzed in more detail in the following section.

### 2.1.3 Scalability, Fault Tolerance, and Consistency

In the design of distributed systems, several fundamental concerns play a crucial role in determining the overall quality, reliability, and performance of the architecture. Among these, scalability, fault tolerance, and consistency are considered core properties, and understanding these concepts is essential for designing systems that are both robust and capable of operating efficiently in real-world environments.

- **Scalability:** Scalability refers to the ability of a system to handle increasing workloads efficiently. Distributed systems achieve scalability primarily through horizontal scaling, which allows services to be replicated and distributed across multiple nodes. This approach enables the system to support higher traffic and larger datasets without relying on a single powerful machine [3].

- **Fault tolerance:** Fault tolerance is the ability of a system to continue operating correctly in the presence of failures. In distributed environments, partial failures are common, meaning that some components may fail while others remain operational [2]. To mitigate these issues, techniques such as replication, retries, and redundancy are commonly employed to improve system resilience and availability.

- **Consistency:** Consistency refers to how data is shared, updated, and synchronized across different nodes in the system. Distributed systems may adopt different consistency models, ranging from strong consistency to eventual consistency, depending on the trade-offs between performance and correctness. These trade-offs are formalized by the CAP theorem, which is represented below.

$$C + A + P \leq 2 \tag{2.1}$$

  The theorem states that: a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance [4].

The interplay between scalability, fault tolerance, and consistency represents one of the central challenges in distributed system design. In practice, improving one of these aspects often comes at the expense of another, requiring careful trade-off decisions based on the specific requirements of the application domain. For this reason, these properties are not considered in isolation but rather as interconnected dimensions that collectively define the behavior and limitations of distributed system architectures.

## 2.2 Microservices Architecture

### 2.2.1 Definition and Principles

Microservices architecture is an architectural style that structures an application as a collection of small, independent services.

A commonly accepted definition describes microservices as *"a suite of small services, each running in its own process and communicating with lightweight mechanisms, often HTTP-based APIs, and independently deployable"* [1].

Each microservice is responsible for a specific business capability and operates as an autonomous unit. Key principles of this architectural style include: loose coupling, high cohesion, and independent deployment. This means that each service can be developed, tested, deployed, and scaled independently without requiring changes to the rest of the system. These characteristics significantly improve development agility and system flexibility.

However, although this architecture improves scalability and maintainability, it also introduces additional complexity in terms of service discovery, monitoring, and especially inter-service communication. As the number of services increases, managing the overall system requires careful orchestration and robust infrastructure support.

### 2.2.2 Advantages and Trade-offs

Microservices provide several advantages and trade-offs that must be carefully considered when designing a distributed system.

- **Advantages:**
  - **Independent scaling:** Microservices enable services to be scaled independently based on their specific load requirements and characteristics. This allows for more efficient resource allocation, as only the components experiencing high demand need to be replicated or scaled.
  - **Increased development velocity:** Since each service is independent, different teams can work on separate microservices simultaneously. This parallel development approach reduces bottlenecks and accelerates the overall development process.

  - **Improved maintainability:** Smaller and well-defined services are generally easier to understand, test, and maintain compared to large monolithic systems.

- **Trade-offs:**
  - **Operational complexity:** Microservices require additional infrastructure for service discovery, monitoring, logging, and orchestration. Managing a large number of distributed services increases system complexity significantly.
  - **Network overhead:** Communication between services occurs over the network, introducing latency and increasing the risk of communication failures.
  - **Distributed system challenges:** Issues such as data consistency, fault tolerance, and debugging become more complex due to the distributed nature of the architecture.

Overall, microservices represent a powerful architectural approach for building scalable and flexible systems. However, their benefits can only be fully realized when the increased operational complexity is properly managed through appropriate tooling, infrastructure, and design practices. So, for this reason, adopting a microservices-based architecture requires a careful evaluation of system requirements and constraints, and also a clear understanding of the trade-offs involved.

### 2.2.3 Service Decomposition Strategies

One of the most critical aspects of microservices architecture is how to decompose a system into different autonomous, independent services. Service decomposition refers to the process of breaking down a monolithic application or a more complex system into smaller, self-contained services, each responsible for a specific business capability. The goal is to ensure that each service has a clear responsibility and can evolve independently from the others, while avoiding the introduction of unnecessary dependencies.

One of the most widely adopted approaches for service decomposition is **Domain-Driven Design (DDD)**, where services are aligned with business domains and bounded contexts [5]. In this approach, the system is analyzed from a business perspective rather than a technical one, and each domain is mapped to its corresponding microservice. A bounded context defines the boundary within which a specific domain model is valid, ensuring that each service encapsulates a well-defined portion of the system's functionality.

However, DDD is not the only possible strategy. Another common strategy is the **decomposition by data ownership**, where each service is responsible for managing its own data and persistence layer. This helps reduce coupling at the database level and supports independent evolution of services, but also requires careful handling of data across services for consistency purposes.

In some cases, systems may also be decomposed using a **technical or layer-based approach**, where services are separated according to technical concerns such as authentication, logging, or API gateways. Although this method can be simpler to implement initially, it often leads to higher coupling and is less aligned with business-driven design principles.

Regardless of the chosen strategy, service decomposition is an iterative process that often requires refinement over time to avoid overly coupled services or excessive communication overhead.

In practice, service decomposition typically involves: identifying business capabilities, analyzing data ownership, and defining service boundaries, minimizing inter-service communication. This process often requires iterative refinement, as initial decompositions may lead to tightly coupled services or to excessive communication overhead.

Improper decomposition can result in bad configurations, the so-called "distributed monoliths," where services are technically separated but still highly dependent on each other, reducing the benefits of the architecture in terms of scalability and maintainability.

## 2.3 Event-Driven Architecture

### 2.3.1 Event-Based Communication Models

Event-Driven Architecture (EDA) is an architectural paradigm in which system components communicate through the production, detection, and consumption of events.

An event is typically defined as a significant change in the state of a system or the occurrence of a relevant action within a predefined domain. Instead of relying on direct, synchronous communication between services, components in an event-driven system use events to which other components can subscribe and react. This constant interaction model enables a reactive flow of information, in which the propagation of changes is driven by events and not by explicit service invocations.

This kind of approach promotes loose coupling between system components, because event producers do not need to be aware of event consumers.

At the same time, and most importantly, it also enables asynchronous communication, the core fundamental feature of this system: improving scalability and responsiveness in distributed environments. However, this systems also introduce additional complexity in terms of event coordination, ordering, and consistency management across all events [6].

### 2.3.2 Publish/Subscribe Pattern

The publish/subscribe pattern is a widely adopted communication model in event-driven systems, designed to enable asynchronous and loosely coupled interactions between all distributed components.

In this model, producers (or publishers) generate events and then send them to an intermediary system, typically referred to as a broker or messaging infrastructure. The consumers (or subscribers), on the other hand, express interest in a specific kind of events and receive only the messages that match their subscription types. This type of separation is managed by the broker, which is also responsible for routing events to the appropriate consumers.

A key characteristic of this pattern is that producers and consumers are completely decoupled from each other. So, they do not need to be aware of each other's existence, their respective identities, or other implementation details. This kind of decoupling mechanism significantly reduces system dependencies and improves modularity. Using this pattern we can add, remove or modifies components without affecting the overall system structure [7].

From a system design perspective, the publish/subscribe pattern also enhances scalability and flexibility. Since communication is asynchronous, producers are not blocked by consumers, and workloads can be distributed more efficiently between services. Additionally, the system can easily accommodate new consumers without requiring changes to existing producers, enabling independent evolution of components. But this model also introduces challenges such as: message ordering, delivery guarantees, and overall an increased complexity in debugging.

### 2.3.3 Benefits and Challenges

Event-driven systems offer several benefits, but also raise new kinds of challenges that must be considered when designing distributed architectures. Here are some of the main strengths and weaknesses of this architecture.

- **Benefits:**
  - **Improved scalability:** Event-driven systems can handle large numbers of events efficiently, as components operate asynchronously and do not block each other.
  - **Decoupling:** Producers and consumers are loosely coupled, meaning they do not need to be aware of each other's existence or implementation details. This improves modularity and system flexibility.
  - **Increased responsiveness:** Systems can react to events in real time, enabling faster processing and improved user experience, especially in dynamic environments.
- **Challenges:**
  - **Message ordering and duplication:** Events may arrive out of order or be delivered multiple times, requiring careful handling to ensure correct system behavior.
  - **Increased debugging complexity:** Asynchronous execution makes it more difficult to trace execution flow and identify the source of issues.
  - **Data consistency issues:** Ensuring a consistent state across distributed components is more complex due to eventual consistency models and asynchronous communication.

We can conclude by saying that this kind of system represents a powerful paradigm for building scalable and flexible distributed architectures. But their adoption also requires careful design decisions and a robust supporting infrastructure to fully leverage their benefits while mitigating their inherent challenges.

## 2.4 Message Queues and AWS SQS

### 2.4.1 Overview of Message Brokers

Message brokers are middleware systems that facilitate communication between distributed components.

They act as intermediaries that receive, store, and deliver messages between producers and consumers. This decouples system components and enables asynchronous communication [8]. Common examples of message

brokers include Apache Kafka, RabbitMQ, Apache ActiveMQ, and Amazon Simple Queue Service (Amazon SQS), which are widely used in microservices and event-driven architectures.

### 2.4.2 Queue-Based Communication

In queue-based communication, messages are sent to a queue and processed by one or more consumers.

Queues provide buffering, allowing producers and consumers to operate at different speeds. They also improve reliability by storing messages until they are successfully processed. If a consumer is temporarily unavailable or overloaded, messages remain in the queue and can be processed later, preventing data loss and improving fault tolerance.

Another important advantage of queue-based systems is decoupling. Producers do not need to know the identity, availability, or implementation details of consumers; they only send messages to the queue. This simplifies the design of distributed applications and allows components to evolve independently.

Queues can also support load balancing by distributing messages among multiple consumer instances. This enables horizontal scalability, since additional consumers can be added to handle higher workloads. Furthermore, many message brokers provide mechanisms such as acknowledgments, retries, and dead-letter queues to guarantee reliable delivery and error handling.

This model is widely used in distributed systems to increase resilience, scalability, and asynchronous processing. Typical use cases include background job execution, order processing systems, notification services, log processing pipelines, and communication between microservices.

### 2.4.3 Characteristics of Amazon SQS

Amazon Simple Queue Service (SQS) is a fully managed message queuing service provided by *Amazon Web Services*.

SQS offers high scalability, durability, and availability. Messages are stored redundantly across multiple servers to ensure reliability. The service follows an *at-least-once delivery* model, meaning that messages may be delivered more than once and must therefore be handled idempotently [9].

SQS supports both standard queues, which provide high throughput with possible duplication, and FIFO queues, which guarantee message ordering and exactly-once processing within constraints.

These characteristics make SQS a suitable choice for building scalable and decoupled distributed systems, although they require careful handling of message processing logic.

# Chapter 3

# Problem Statement

## 3.1 Existing System Overview

### 3.1.1 High-Level Architecture of the Current System

The system addressed in this work is the backend architecture of Legenda, a distributed application built using FastAPI and based on an asynchronous, event-driven design. Its purpose is to manage document processing, semantic search, and AI-powered workflows, while ensuring scalability and responsiveness.

The architecture follows a layered approach composed of four layers that work together: the API layer, the service layer, the repository layer, and the data layer. In this architecture, the API layer exposes REST endpoints and handles client requests, while the service layer contains the core business logic and orchestrates all the workflows. Then we have the repository layer that manages the access to databases, and also the data layer that manages persistence and indexing services.

To avoid blocking synchronous API requests, which will slow down operations, the system relies heavily on asynchronous background processing and parallel execution. In this scenario, long-running or resource-intensive operations are delegated to background workers through message queues. Communication between components is achieved using Amazon SQS, which acts as the message broker of the system, using a producer/consumer pattern.

The high-level architecture flow can be summarized as follows:

1. The API receives requests from users or external systems.
2. The API publishes asynchronous jobs to an SQS queue.

3. Dedicated workers consume messages from the queue.
4. Workers execute business logic by invoking internal services.
5. Services interact with databases and storage systems such as PostgreSQL, OpenSearch, Weaviate, and AWS S3.

This design enables asynchronous execution and decouples the request-response lifecycle from expensive background tasks.

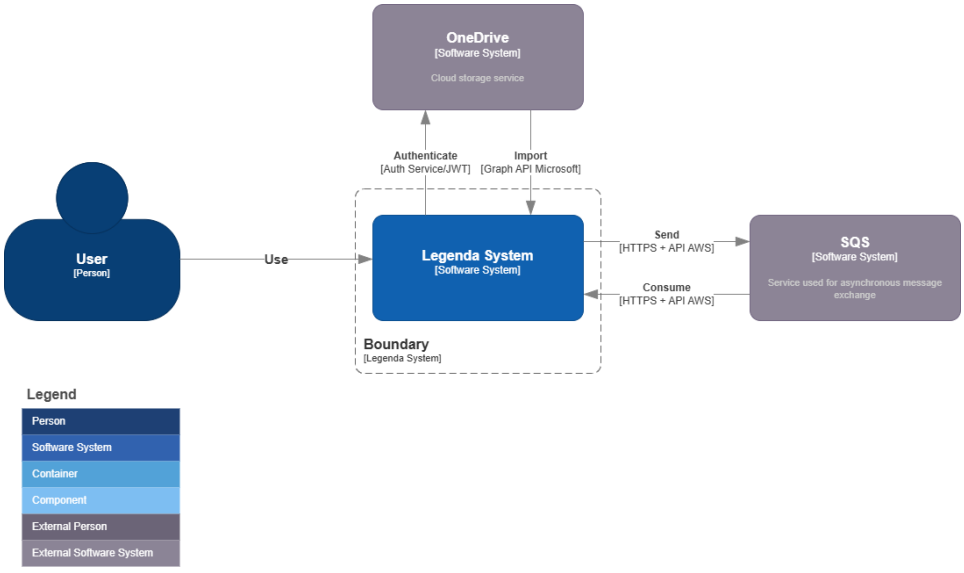


Figure 3.1: System Diagram.

### 3.1.2 Components and Services Involved

The system is composed of several interconnected components and services organized according to a layered architecture model. The adoption of a layered architecture is a common software engineering practice that promotes and grants: separation of concerns, modularity, maintainability, and scalability.

Separation of concerns is considered a fundamental principle in software engineering because it allows developers to isolate functionalities into independent modules with clearly defined responsibilities. According to Snoeck et al. [10], separation of concerns is *"a determining factor of the quality*

*of object-oriented software development"* and provides benefits such as improved adaptability, maintainability, and reuse. Similarly, Tarr et al.[11] state that proper separation of concerns reduces complexity and simplifies software evolution.

The analyze architecture follows this layered approach and is composed of several logical layers and components that collaborate to provide: document management, semantic search, and asynchronous background processing capabilities.

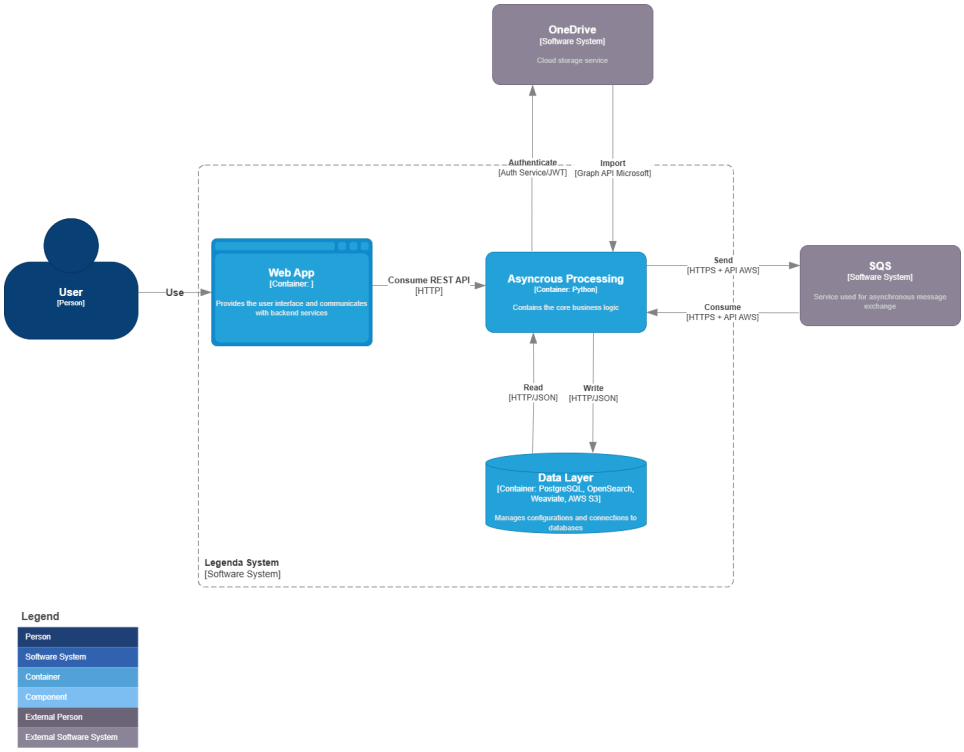


Figure 3.2: Container Diagram.

### API Layer

The API layer is implemented using FastAPI and acts as the main entry point of the application. This layer uses REST endpoints and manages authentication and authorization mechanisms.

Its primary responsibility is handling HTTP communication while delegating business operations to the service layer. This separation ensures that

transport-related concerns remain isolated from the core business logic.

### Service Layer

The service layer contains the core business logic of the application and orchestrates workflows involving: repositories, databases, external services, and asynchronous workers.

Examples of operations handled by this layer include:

- semantic search execution,
- document management,
- folder synchronization,
- notification handling,
- quota tracking.

Several internal services participate in these workflows, including:

- `search_service`,
- `folder_service`,
- `pdf_binary_service`,
- `quota_service`.

The service layer abstracts business processes from infrastructure details, improving modularity and facilitating testing activities.

### Repository and Data Layer

The repository layer abstracts access to the persistence systems used by the application. It encapsulates database queries and storage operations, reducing direct dependencies between business logic and infrastructure technologies.

### Asynchronous Processing Infrastructure

To support long-running and resource-intensive tasks, the system adopts an asynchronous processing infrastructure based on Amazon Simple Queue Service (SQS) and using background workers.

A task should not block the synchronous HTTP request lifecycle, so the API publishes a message to an SQS queue. The primary queue used for generic asynchronous processing is `QUEUE_URL_WORKERS_JOB_RUNNER`.

Messages are serialized as JSON objects and typically contain:

- an `event_name` field identifying the event type,
- payload data required for processing,
- metadata associated with the operation.

SQS acts as a message broker between producers and consumers, ensuring reliable message delivery, buffering, and asynchronous communication between distributed components.

#### Job Runner Worker

The Job Runner Worker is the central asynchronous processing component of the system. Its responsibility is to consume messages from SQS, interpret event payloads, and invoke the appropriate business services.

The worker operates in two execution modes:

- AWS Lambda mode, used in production environments,
- Polling mode, used during local development.

The implementation relies on a centralized dispatcher based on a long `if/elif/else` structure. After parsing a JSON message and extracting the `event_name` field, the worker manually routes the message to the corresponding service function.

Although functional, this implementation introduces strong coupling between the worker and the service layer, making the architecture progressively harder to maintain and evolve as the number of supported events increases.

## 3.2 SQS-Based Processing Model

### 3.2.1 Message Structure

The asynchronous processing model is based on message exchange through Amazon SQS.

When an operation requires background execution, the API or another service publishes a message to the SQS queue using `send_message()`. Messages are encoded as JSON strings and include a `event_name` field used to identify the type of operation to execute.

A typical message structure includes:

- the event identifier,
- resource identifiers,
- operation-specific payload data,
- optional metadata.

Once the message is stored in the queue, the Job Runner Worker retrieves it using either Lambda event triggers or queue polling.

### 3.2.2 Application Workflow

After fetching a message from the Amazon SQS queue, the Job Runner Worker executes a sequence of operations to process the event and execute the required business logic. The workflow can be divided into the following phases:

1. **Parsing the JSON Body**

   At this stage, the worker extracts all the information contained inside the payload, including event identifiers, metadata, and operation-specific parameters. Since messages are represented as generic JSON objects, no strict schema validation is applied during parsing.

   This approach provides flexibility but also introduces risks. Missing fields, malformed payloads, or invalid data types may not be detected immediately and can cause runtime errors during later processing stages.

2. **Extracting the `event_name`**

   After parsing the payload, the worker extracts the `event_name` field from the message body. This field represents the type of asynchronous operation that must be executed and acts as the main routing mechanism of the architecture.

   The `event_name` determines which business workflow will be triggered by the worker.

   Each event corresponds to a different asynchronous task and may require interactions with different services, databases, or external systems.

3. **Routing the Message to the Corresponding Handler**

   Once the event type has been identified, the worker routes the message to the appropriate handler using a centralized dispatcher implemented through a large `if/elif/else` structure.

   The dispatcher manually compares the value of `event_name` against all supported event types and invokes the corresponding service function. For example:

   - search events are routed to the `search_service`,
   - document movement events are routed to the `pdf_binary_service`,
   - folder-related events are routed to the `folder_service`

   This routing mechanism centralizes event management inside a single component. While functional, this design introduces strong coupling between the worker and the business services because every new event requires modifying the dispatcher logic directly.

4. **Executing the Business Logic**

   After routing the event, the selected service executes the required business logic associated with the message. During execution, services may interact with several infrastructure components and also trigger additional asynchronous events, invoking external APIs and others AI-related services.

5. **Deleting the Message from the Queue**

   If the business logic execution completes successfully, the worker acknowledges the message and removes it from the SQS queue.

   Message deletion is a crucial step because it prevents the same event from being processed multiple times. The deletion operation confirms that the asynchronous task has been completed correctly.

   If an error occurs before this phase, the message is not removed from the queue. Instead, after the visibility timeout expires, the message becomes available again for processing. This retry mechanism improves fault tolerance and reliability by allowing transient failures to be handled automatically.

This workflow represents the core execution model of the current asynchronous processing architecture and highlights both its operational behavior, but also its architectural limitations.

## 3.3 Identified Issues of the Current System

### 3.3.1 Lack of Type Safety

One of the major limitations of the current architecture is the absence of strict typing in message handling.

Messages are represented as untyped JSON dictionaries and parsed manually using `json.loads()`. Event payload fields are extracted directly from dictionaries without formal schema validation.

As a consequence:

- missing fields may cause runtime exceptions,
- invalid payload formats are detected only during execution,
- developers must rely on implicit assumptions about message structure,
- IDE support and static analysis capabilities are limited.

This makes integrations fragile and increases the probability of runtime failures caused by malformed messages.

### 3.3.2 Weak Schema Enforcement

The current system lacks centralized schema enforcement for asynchronous events.

Each producer constructs JSON payloads independently, while consumers manually interpret message fields. Since no shared contract exists between producers and consumers, inconsistencies may emerge over time.

Examples of potential issues include:

- renamed fields,
- missing attributes,
- incompatible payload structures,
- inconsistent data types.

Without strict validation, these inconsistencies may remain undetected until the message reaches production environments. This weakens reliability and complicates debugging activities.

### 3.3.3 Tight Coupling in Message Handling Logic

Another major issue is the tight coupling introduced by the monolithic dispatcher.

The worker directly imports many service modules and explicitly routes events through a large `if/else` chain. This design creates several problems:

- every new event requires modification of the central dispatcher,
- the worker becomes increasingly complex as the number of events grows,
- business logic routing is centralized in a single file,
- event handlers cannot be easily isolated or tested independently.

Furthermore, the architecture relies on implicit assumptions regarding event payloads and service dependencies. These hidden dependencies reduce modularity and make the system harder to evolve.

## 3.4 Impact on System Quality

### 3.4.1 Maintainability

The current architecture negatively affects maintainability.

The centralized dispatcher creates a monolithic structure where adding or modifying an event requires editing the core worker logic. As the number of supported events increases, the file becomes harder to navigate and understand.

Additionally, the lack of typed schemas makes refactoring risky because developers cannot rely on compile-time validation or automatic consistency checks. This increases the maintenance cost and reduces development velocity.

### 3.4.2 Scalability

Although the use of Amazon SQS enables asynchronous processing and horizontal scaling, the current implementation introduces architectural bottlenecks.

The Job Runner Worker concentrates routing responsibilities inside a single dispatcher, making the component increasingly difficult to scale from a software engineering perspective. As the number of event types grows, complexity grows proportionally.

Moreover, tightly coupled event handling logic limits the possibility of independently evolving or distributing consumers across multiple specialized workers.

### 3.4.3 Reliability

The absence of strict schema validation and typed event contracts reduces overall system reliability.

Malformed or incomplete messages may propagate through the system until runtime errors occur. Since validation is mostly manual, failures may happen deep inside the business logic, complicating debugging and recovery.

In addition, centralized routing increases the risk that errors in the dispatcher affect unrelated event processing flows. Implicit dependencies and inconsistent payload structures also contribute to fragile integrations between producers and consumers.

As a result, failures can propagate across components and negatively impact the stability of the asynchronous processing pipeline.

# Chapter 4

# Proposed Solution: Typed Event-Bus Registry

## 4.1 Design Goals

The proposed solution introduces a Typed Event-Bus Registry architecture aimed at addressing the main limitations identified in the current asynchronous processing system. In this particular scenario, the solution focuses on improving: type safety, extensibility, decoupling, and observability within the event-driven infrastructure. Recent studies on Event-Driven Architectures (EDA) highlight how asynchronous publish/subscribe communication enables loose coupling, scalability, and resilience in distributed systems. In particular, typed event-bus solutions improve type safety and extensibility by defining structured event contracts between producers and consumers, while also enhancing observability and maintainability across the infrastructure [12].

### 4.1.1 Type Safety

One of the primary goals of the proposed architecture is to introduce strong type safety for asynchronous event communication.

In the current implementation, messages are exchanged as typed JSON objects and manually parsed at runtime. This approach increases the likelihood of runtime failures due to malformed payloads, missing fields, or inconsistent data structures.

To mitigate these issues, the new architecture adopts a typed event schema based on Pydantic models. Each event is represented by a strongly

typed class defining:

- the event structure,
- required attributes,
- optional metadata,
- validation constraints.

This approach enables automatic validation during serialization and deserialization phases, reducing the risk of invalid payload propagation across the system.

Additionally, the adoption of typed schemas significantly improves the overall development experience and software quality throughout the system lifecycle.

From a development perspective, strong typing enables modern IDEs to provide advanced support features such as:

- automatic code completion for event attributes and methods,
- real-time detection of type mismatches,
- navigation between event definitions and their corresponding handlers,
- improved static analysis during implementation and refactoring phases.

These capabilities reduce development errors and allow developers to identify inconsistencies at design time rather than at runtime. This is particularly important in distributed asynchronous systems, where runtime failures may be harder to reproduce and debug due to the decoupled nature of event processing.

Typed schemas also improve maintainability during system evolution. Since event contracts are explicitly defined and shared across producers and consumers, modifications to event structures become immediately visible to the development environment and to static validation tools. This allows breaking changes to be detected early, reducing the risk of introducing hidden incompatibilities between distributed components.

Furthermore, explicit type definitions improve code readability and documentation quality. Event structures become self-descriptive artifacts that clearly communicate the expected data model, reducing ambiguity and simplifying onboarding for new developers working on the system.

By enforcing strict contracts between producers and consumers, the system becomes more predictable and reliable. The validation layer guarantees

that only structurally correct messages can enter the processing pipeline, preventing malformed payloads from propagating through the asynchronous infrastructure and causing failures deep inside business logic execution.

### 4.1.2 Extensibility

Another important objective of the proposed solution is improving extensibility. Extensibility is the capability of a software system to accommodate new functionalities, components, or behaviors with minimal modifications to the existing codebase. In distributed and event-driven systems, extensibility is considered a fundamental quality attribute because it enables systems to evolve incrementally while preserving maintainability and reducing architectural complexity [12].

In modern asynchronous infrastructures, extensibility becomes particularly important due to the continuous introduction of new workflows, event types, and integration requirements. Architectures that tightly couple business logic with centralized dispatching mechanisms often become difficult to maintain and evolve.

The existing architecture relies on a centralized dispatcher implemented through a large `if/else` structure. Every new event requires direct modification of the core worker logic, progressively increasing system complexity.

The Typed Event-Bus Registry removes this limitation by introducing a modular registration mechanism where handlers can be dynamically associated with event types.

This design allows developers to:

- introduces new event types without modifying the central dispatcher,
- register new handlers independently,
- extends the system incrementally,
- isolate business logic into dedicated components.

As a consequence, the architecture becomes easier to evolve and better suited for large-scale distributed systems where the number of asynchronous workflows may grow significantly over time.

### 4.1.3 Decoupling

Reducing coupling between components represents another central design goal.

In the current implementation, the Job Runner Worker directly imports multiple services and explicitly routes messages to specific functions. This creates strong dependencies between infrastructure and business logic.

The proposed Event-Bus Registry introduces an abstraction layer between event producers and consumers. Producers only emit typed events, while consumers register themselves independently inside the registry.

This approach improves loose coupling because:

- producers do not need knowledge of consumers,
- handlers can evolve independently,
- infrastructure concerns are separated from business logic,
- services become more modular and reusable.

With this implementation, the architecture aligns more closely with the principles of event-driven and microservices-based systems discussed in the previous chapters.

### 4.1.4 Observability

The final design goal concerns observability and traceability of asynchronous workflows.

Asynchronous distributed systems are often difficult to debug because execution flows are fragmented across multiple services and events. In the current architecture, tracing event propagation and identifying failures can become complex.

The proposed solution improves observability by centralizing event metadata management and standardizing event structures. Each event can include additional contextual information such as timestamps, correlation identifiers, event versions, and processing-related metadata.

This information enables several important observability capabilities that are fundamental in distributed asynchronous architectures.

- **Improved logging:** Standardized metadata enables consistent and more readable logs across distributed components.
- **Distributed tracing:** Correlation identifiers support end-to-end tracing of asynchronous workflows across services.
- **Easier debugging:** Typed events simplify failure detection and reduce debugging complexity.

- **Monitoring of asynchronous workflows:** Event metadata enables real-time monitoring of processing states, retries, and failures.

Overall, observability becomes a core architectural characteristic rather than an additional operational concern. The standardized management of events and metadata provides greater visibility into the internal behavior of the system and facilitates operational control over distributed asynchronous workflows.

As a result, the system becomes more transparent and operationally manageable, especially in production environments where large numbers of events are processed concurrently.

## 4.2 Event Modeling

### 4.2.1 Definition of Domain Events

A fundamental aspect of the proposed architecture is the formal definition of domain events.

In event-driven systems, an event represents a significant change in the state of the application or the occurrence of a relevant business action. According to Domain-Driven Design principles, a domain event can be defined as *"something that happened in the domain"* [13]. Domain events are therefore used to communicate meaningful state changes asynchronously between distributed components while preserving loose coupling.

Within the proposed architecture, domain events are modeled as explicit representations of business operations occurring inside the system. Each event corresponds to a meaningful action within the application domain and encapsulates all the information required for asynchronous processing.

Examples of domain events in the analyzed system include:

- document upload completion,
- semantic search execution requests,
- folder synchronization updates,
- notification reception events,

Instead of representing these operations as generic JSON dictionaries, the new architecture models them as dedicated typed entities and, by formalizing domain events, the architecture improves consistency across the asynchronous communication layer, reducing ambiguity during system evolution.

### 4.2.2 Schema Design (Pydantic Models)

To guarantee strict validation and type safety, the proposed solution adopts typed event schemas implemented using Pydantic models.

Pydantic is a Python data validation library based on type annotations. It enables automatic validation, serialization, and parsing of structured data while preserving strong typing guarantees.

Each event is represented through a dedicated schema class defining:

- event attributes,
- data types,
- validation rules,
- optional metadata fields.

A simplified example of a typed event model is shown below:

```
class SearchExecutionStartedEvent(BaseModel):
    event_name: str
    search_id: UUID
    user_id: UUID
    timestamp: datetime
```

When an event is produced or consumed, Pydantic automatically validates:

- required fields,
- field types,
- missing attributes,
- invalid payload structures.

If validation fails, the message can be rejected before reaching the business logic layer, preventing runtime failures caused by malformed payloads.

The adoption of typed schemas also improves: code readability, IDE integration, static analysis, automatic documentation generation.

Moreover, schema definitions become centralized and reusable across producers and consumers, ensuring consistency throughout the distributed system.

## 4.3 Event-Bus Architecture

### 4.3.1 Central Registry Concept

The core component of the proposed solution is the **Typed Event-Bus Registry**, a centralized software component responsible for managing: the registration, the resolution, and the dispatching of domain events inside the asynchronous processing infrastructure.

In software architecture, an *event bus* is commonly defined as a communication mechanism that enables different components of a system to exchange events in a decoupled manner. Rather than directly invoking each other, components publish events to the bus, while interested consumers subscribe to specific event types and react accordingly. This architectural pattern is widely adopted in event-driven systems because it promotes loose coupling, modularity, and asynchronous communication between distributed services.

Within the proposed architecture, the Event-Bus does not replace Amazon SQS as the messaging infrastructure. Instead, it operates at the application layer as an internal orchestration and routing mechanism responsible for:

- maintaining associations between event types and handlers,
- validating and resolving typed events,
- dispatching events to the appropriate consumers,
- abstracting routing logic from the worker implementation.

This abstraction removes the need for direct dependencies between producers and consumers. Services only need to publish events to the bus without knowing which component will process them. Similarly, handlers can be registered independently without requiring modifications to the core worker logic.

The central registry is therefore responsible for storing the mapping between event schemas and their corresponding handlers. Unlike the previous implementation based on a monolithic `if/else` dispatcher, routing decisions are no longer hardcoded but dynamically resolved at runtime.

When a worker receives a message from Amazon SQS, the payload is first deserialized into a typed event model and then submitted to the Event-Bus. The registry identifies the corresponding handler associated with the event type and dynamically invokes the appropriate business logic.

This architecture introduces a more modular and extensible execution model where event management responsibilities are centralized while still

preserving loose coupling between components. As a consequence, the system becomes easier to maintain, extend, and evolve over time, especially as the number of asynchronous workflows increases.

### 4.3.2 Event Dispatching Mechanism

The event dispatching mechanism is responsible for routing incoming events to the appropriate handlers inside the asynchronous processing infrastructure.

In the proposed architecture, dispatching is no longer based on hardcoded conditional statements but on a dynamic resolution process managed by the Typed Event-Bus Registry. This mechanism introduces an abstraction layer between message consumption and business logic execution, allowing the worker to remain generic and independent from specific application workflows.

When a message is received from Amazon SQS, the worker performs a sequence of operations that transforms the raw message into a validated domain event and dynamically routes it to the correct consumer.

1. **Message deserialization**

   The worker first retrieves the raw JSON payload from the SQS queue and deserializes it into an internal event representation. Unlike the previous implementation, where messages were treated as generic dictionaries, the new architecture attempts to map the payload to a specific typed event schema.

2. **Schema validation**

   After deserialization, the payload is validated against the corresponding typed model using Pydantic schemas. During this phase, the system verifies:

   - the presence of required fields,
   - data type correctness,
   - structural consistency of the payload,
   - compatibility with the expected event schema.

   If validation fails, the event can be rejected immediately before reaching the business logic layer. This prevents malformed or inconsistent messages from propagating through the distributed system.

3. **Event type identification**

   Once validation succeeds, the worker identifies the event type associated with the payload. The event type acts as the primary routing key used by the Event-Bus Registry to determine which handler must process the event.

   Since events are represented through explicit schema classes, event identification becomes more reliable and less dependent on manually interpreted string values.

4. **Handler resolution through the registry**

   The validated event is then submitted to the Event-Bus Registry. The registry maintains an internal mapping between event types and their corresponding handlers.

   Instead of evaluating multiple conditional branches, the registry dynamically resolves the appropriate consumer associated with the event. This mechanism significantly simplifies the worker implementation and removes centralized routing logic from the infrastructure layer.

5. **Dynamic handler invocation**

   After the correct handler has been resolved, the Event-Bus invokes the corresponding processing logic dynamically. Each handler encapsulates a specific business workflow and operates independently from the worker itself.

   Handlers may interact with:

   - internal application services,
   - databases,
   - storage systems,
   - external APIs,
   - additional asynchronous workflows.

   Once processing completes successfully, the worker acknowledges the message and removes it from the queue.

Unlike the previous implementation, the worker no longer contains explicit routing logic for each event type. The infrastructure component becomes significantly more generic, modular, and maintainable. Adding support for new asynchronous workflows no longer requires modifications to the

central dispatcher, since new handlers can simply register themselves inside the Event-Bus Registry.

This architecture improves separation of concerns by isolating infrastructure responsibilities, event validation, routing logic, and business processing into distinct and dedicated components.

As a consequence, the event processing pipeline becomes easier to extend, test, and evolve over time while preserving loose coupling between distributed components.

### 4.3.3 Handler Registration and Resolution

Handlers are dynamically registered inside the Event-Bus Registry.

Each handler declares:

- the event type it supports,
- the processing logic associated with the event.

Registration may occur during application startup through decorators, configuration modules, or explicit initialization procedures.

A simplified example is shown below:

```
@event_handler(SearchExecutionStartedEvent)
class SearchExecutionHandler:

    async def handle(self, event):
        ...
```

During initialization, the registry automatically associates: the event schema and the corresponding handler implementation.

When an event is dispatched, the registry resolves the appropriate handler dynamically without requiring modifications to the worker infrastructure.

Moreover, individual handlers can be developed and maintained independently, reducing complexity and improving separation of concerns.

## 4.4 Integration with Existing Infrastructure

### 4.4.1 Event Processing Workflow within the Existing SQS Infrastructure

The proposed Event-Bus Registry architecture is designed to integrate with the existing Amazon SQS infrastructure without replacing it.

Amazon SQS continues to operate as the message broker responsible for:

- asynchronous communication,
- message buffering,
- fault tolerance,
- message delivery.

The main architectural change concerns the internal processing model used by consumers after retrieving messages from the queue.

Instead of manually parsing generic JSON payloads and routing events through conditional logic, the new architecture introduces typed event deserialization, schema validation, and dynamic handler resolution through the registry.

The overall communication flow remains compatible with the existing infrastructure, while introducing additional validation and abstraction layers inside the asynchronous processing pipeline.

1. **Event production and publication to SQS**

   When an asynchronous operation must be executed, a producer component generates a domain event representing a specific business action or state change. Instead of constructing generic and unstructured JSON payloads, the event is instantiated using a typed schema model that validates the payload before publication.

   Once validated, the event is serialized into JSON format and published to the appropriate Amazon SQS queue. At this stage, SQS continues to operate as the distributed messaging infrastructure responsible for reliable message delivery, buffering, and asynchronous communication between components.

2. **Message consumption by workers**

   Dedicated background workers continuously consume messages from the SQS queue through polling mechanisms or AWS Lambda event triggers. The worker acts as the entry point of the asynchronous processing pipeline and is responsible for retrieving the serialized event payload from the queue.

   Unlike the previous implementation, the worker no longer contains explicit business routing logic or hardcoded event dispatching structures.

3. **Typed event deserialization and validation**

   After receiving a message, the worker deserializes the JSON payload into the corresponding typed event model. During this phase, schema validation is automatically performed using Pydantic models.

   The validation layer verifies:

   - required fields,
   - data types,
   - payload consistency,
   - schema compatibility.

   Invalid or malformed messages can therefore be detected immediately before reaching the business logic layer, reducing runtime failures and improving system reliability.

4. **Handler resolution through the Event-Bus Registry**

   Once the event has been successfully validated, it is submitted to the Event-Bus Registry. The registry dynamically resolves the handler associated with the event type by consulting its internal mapping between event schemas and registered consumers.

   This mechanism completely replaces the centralized `if/elif/else` dispatcher previously used by the worker. Routing decisions are therefore abstracted from the infrastructure layer and managed dynamically at runtime.

5. **Business logic execution**

   After the appropriate handler has been resolved, the Event-Bus invokes the corresponding business logic asynchronously. Handlers may interact with:

   - internal services,
   - databases,
   - storage systems,
   - external APIs,
   - additional asynchronous workflows.

   Once processing completes successfully, the worker acknowledges the message and removes it from the SQS queue. In case of failures, standard SQS retry and visibility timeout mechanisms continue to guarantee fault tolerance and reliable message delivery.

This execution flow preserves the operational advantages of Amazon SQS while introducing stronger typing, modular event dispatching, and improved separation of concerns at the application level.

### 4.4.2 Backward Compatibility

Backward compatibility represents a critical requirement for adopting the new architecture incrementally without disrupting existing services.

The proposed solution is designed to coexist temporarily with the current dispatcher-based implementation during the migration phase.

Legacy events may continue to be processed using the existing routing logic, while new events progressively adopt the typed Event-Bus mechanism.

A possible hybrid approach can also be introduced, where legacy events continue using the traditional dispatcher while new typed events adopt the registry-based architecture.

This gradual migration strategy provides several advantages:

- reduced migration risk,
- incremental adoption,
- minimal disruption to production environments,
- preservation of existing integrations.

Additionally, compatibility adapters may be introduced to transform legacy JSON payloads into typed event schemas when required.

By avoiding a complete architectural rewrite, the system can evolve progressively while maintaining operational continuity and minimizing deployment risks.

# Chapter 5

# Implementation of the Proposed Architecture

## 5.1 System Design

After defining the theoretical foundations of distributed systems, microservices, and event-driven architectures, and after analyzing the limitations of the existing implementation, the next step consists in describing the practical implementation of the proposed Typed Event-Bus Registry architecture.

The implementation was designed to integrate directly with the existing FastAPI and Amazon SQS infrastructure already used by the system. Rather than replacing the asynchronous processing model, the proposed solution introduces an additional abstraction layer responsible for managing typed events, centralizing event registration, dynamically resolving handlers, validating schemas, and decoupling message processing logic from the worker infrastructure.

The main objective of the implementation is to remove the limitations introduced by the previous dispatcher-based architecture, where all asynchronous workflows were managed through a large centralized `if/elif/else` structure.

The new architecture introduces three main components:

- typed event schemas,
- the Event-Bus Registry,
- consumer handlers.

The interaction between these components enables a more modular and extensible asynchronous processing pipeline.

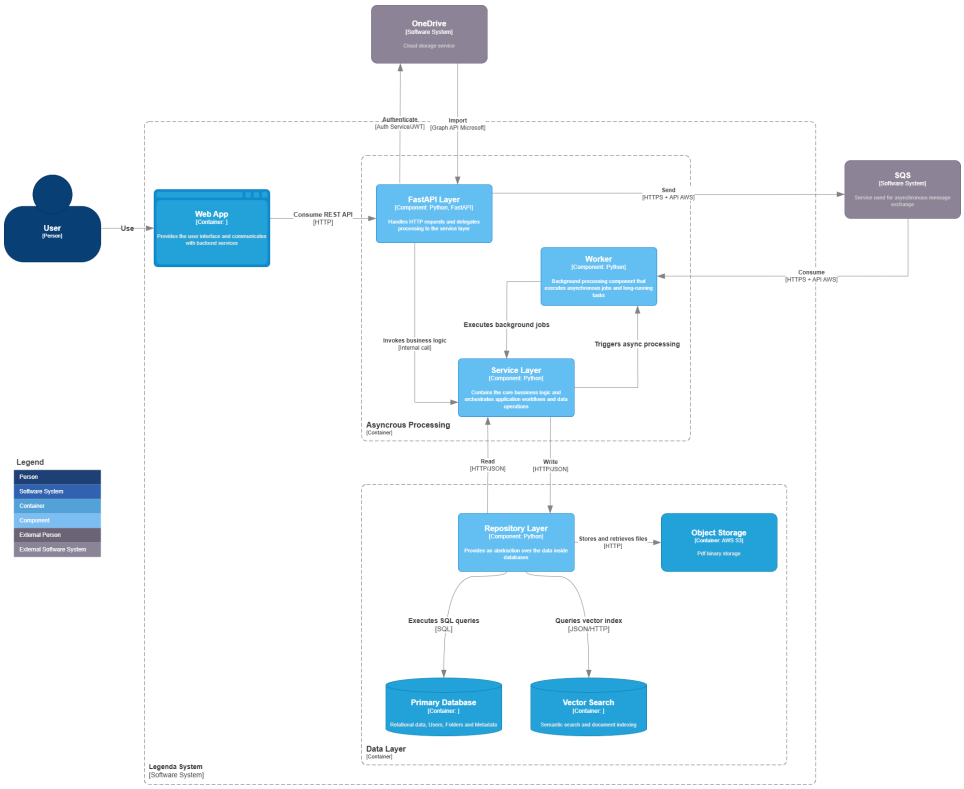


Figure 5.1: Component Diagram.

## 5.2 Event Definitions

A fundamental aspect of the proposed architecture is the introduction of typed event definitions.

Instead of representing asynchronous messages as generic dictionaries, every event is modeled through explicit Pydantic schemas. This approach introduces a formal and centralized definition of the data exchanged between distributed components, improving consistency and reliability throughout the asynchronous processing pipeline.

Each event is represented as a dedicated class inheriting from a common `BaseEvent` abstraction. The use of typed schemas enables automatic valida-

tion of incoming payloads, ensuring that only structurally correct messages can be processed by the system. In addition, explicit type definitions improve readability, maintainability, and integration with static analysis tools and modern IDEs.

### 5.2.1 BaseEvent Abstraction

All asynchronous events inherit from a common base abstraction called `BaseEvent`.

The implementation is shown below:

```
class BaseEvent(BaseModel):
    event_name: str
```

The purpose of this abstraction is to provide a shared structure for all domain events inside the system.

All concrete event implementations inherit from this base schema and extend it with domain-specific attributes.

In addition to `BaseEvent`, the implementation also introduces a reusable `UserSchema` model:

```
class UserSchema(BaseModel):
    tenant_id: int
    username: str
    use_vector_db: bool = True
    legenda_user_id: int | None = None
    tenant_name: str | None = None
```

This schema centralizes user-related information shared across multiple asynchronous workflows.

### 5.2.2 Typed Event Classes

The architecture defines several typed event classes corresponding to different business operations.

Each event explicitly describes:

- required attributes,
- supported data types,
- payload structure,

- business context.

One example is the `SearchExecutionStartedEvent`:

```
class SearchExecutionStartedEvent(BaseEvent):
     event_name: Literal['search-execution-started-event']
     user: UserSchema
     search_query_id: int
     search_payload: SearchInSchema
```

This event is emitted whenever a semantic search execution starts.

The schema guarantees that the event contains a valid user structure, that the search identifier is correctly provided, and that the payload complies with the expected schema definition. As a consequence, malformed or incomplete messages can be detected immediately during validation, preventing invalid data from propagating through the asynchronous processing pipeline.

To support dynamic validation, all events are grouped inside the `JobRunnerEvent` union:

```
JobRunnerEvent = Union[
    SearchExecutionStartedEvent,
    ReportExecutionStartedEvent,
    ...
]
```

This union enables Pydantic to automatically identify and validate the correct schema during deserialization.

The parsing process is implemented through the following function:

```
def parse_event(raw: dict) -> BaseEvent:
    return TypeAdapter(JobRunnerEvent).validate_python(raw)
```

This mechanism represents one of the core improvements introduced by the proposed architecture because it eliminates manual payload parsing and introduces centralized schema validation.

## 5.3 Event Registry

The core component of the proposed architecture is the Event-Bus Registry. The registry acts as an internal orchestration layer responsible for:

- **Storing associations between events and handlers.** This mechanism maintains a centralized mapping between each event type and its corresponding handler implementation, ensuring a consistent binding between event definitions and execution logic across the entire system lifecycle. It provides a single source of truth for event-handler relationships, improving maintainability and traceability.
- **Resolving consumers dynamically.** At runtime, the registry determines the appropriate handler based on the event type, avoiding any reliance on static conditional structures such as `if/else` chains. This dynamic resolution mechanism allows the system to remain extensible, as new event types can be introduced without modifying the core dispatching logic.
- **Dispatching validated events.** After successful schema validation, events are forwarded to their corresponding handlers for execution within the business layer. This ensures that only structurally valid and type-safe events are propagated through the processing pipeline, reducing the risk of runtime inconsistencies.
- **Abstracting routing logic from the worker.** The responsibility of routing events is entirely delegated to the registry, allowing the worker to remain generic and focused solely on message consumption and lifecycle management. This separation of concerns reduces coupling between infrastructure and business logic and improves the overall modularity of the system.

### 5.3.1 Registry Structure

The Event-Bus implementation is based on a centralized registry structure:

```
class EventBus:
    def __init__(self):
        self._handlers: Dict[str, Callable] = {}
```

The registry internally maintains a dictionary where:

- the key represents the `event_name`,
- the value represents the associated handler function.

This structure replaces the previous `if/elif/else` dispatcher used by the Job Runner Worker.

The registration mechanism is implemented through the `register_handler()` method:

```
def register_handler(self, event_name: str, handler: Callable):
    if event_name in self._handlers:
        raise ValueError(...)

    self._handlers[event_name] = handler
```

The implementation also prevents duplicate registrations, guaranteeing that each event type is associated with only one handler.

### 5.3.2 Handler Mapping

The connection between events and handlers is implemented through a decorator-based registration mechanism.

The `consumer()` decorator automatically associates an event schema with its corresponding consumer function.

The implementation is shown below:

```
def consumer(event_cls: type[BaseEvent]):
    def decorator(func: Callable):
        event_name = event_cls.model_fields["event_name"].default
        event_bus.register_handler(event_name, func)
        return func
    return decorator
```

Using decorators significantly simplifies handler registration and improves code readability.

A concrete consumer implementation is shown below:

```
@consumer(events.SearchExecutionStartedEvent)
async def handle_search_started(event):
    ...
```

During application startup:

1. The decorator extracts the `event_name` from the event class in order to uniquely identify the type of event that the handler is intended to process. This ensures that each handler is explicitly bound to a well-defined event schema.

2. The handler is registered automatically through the decorator mechanism, which links the handler function to the corresponding event type without requiring manual configuration. This approach reduces boilerplate code and minimizes the risk of inconsistencies during registration.

3. The Event-Bus stores the mapping internally, building a centralized registry that associates each `event_name` with its corresponding handler. This internal state is then used at runtime to enable dynamic event dispatching within the processing pipeline.

## 5.4 Message Processing Pipeline

The asynchronous processing pipeline represents the operational flow used by the system to generate, transmit, validate, and process events.

The proposed implementation preserves the existing Amazon SQS infrastructure while introducing typed validation and dynamic dispatching.

### 5.4.1 Event Creation at API Level

When an asynchronous operation must be executed, the API creates a typed event instance instead of manually constructing generic JSON payloads.

An event is instantiated using the corresponding schema:

```
SearchExecutionStartedEvent(
    user=current_user,
    search_query_id=query_id,
    search_payload=payload
)
```

During event creation, Pydantic automatically validates:

- required fields,
- field types,
- nested schemas,
- payload consistency.

If validation succeeds, the event is serialized and published to Amazon SQS.

This approach guarantees that only structurally valid events enter the asynchronous pipeline.

### 5.4.2 Serialization and Deserialization

After creation, the event is serialized into JSON format before being sent to the queue.

Serialization allows typed Python objects to be transformed into transportable message payloads compatible with Amazon SQS.

Once the worker retrieves the message from the queue, the deserialization phase begins.

The Event-Bus executes the following operations:

1. extraction of the JSON payload,
2. identification of the `event_name`,
3. schema resolution through `JobRunnerEvent`,
4. automatic validation using Pydantic,
5. conversion into a typed event instance.

The implementation is performed through:

```
event: BaseEvent = parse_event(message_body)
```

This mechanism eliminates the need for manual field extraction and reduces runtime parsing errors.

### 5.4.3 Consumer-Side Handling

After successful validation, the Event-Bus resolves the correct handler dynamically.

The dispatching process is implemented inside the `dispatch()` method:

```
handler = self._handlers.get(event.event_name)
```

If a valid handler exists, the Event-Bus invokes the corresponding asynchronous consumer:

```
await handler(event)
```

Each consumer is responsible for a specific business workflow.

For example, the `handle_search_started()` consumer executes semantic search operations:

```
@consumer(events.SearchExecutionStartedEvent)
async def handle_search_started(event):
    await get_search_service().execute_search(...)
```

## 5.5 Error Handling and Validation

One of the primary goals of the proposed architecture is improving reliability through centralized validation and structured error handling.

The new implementation introduces validation mechanisms at multiple stages of the asynchronous processing pipeline.

### 5.5.1 Pydantic Validation

Pydantic validation is performed automatically during:

- event creation,
- event deserialization,
- nested schema parsing.

The validation process verifies the presence of all required fields, the correctness of field types, the overall schema compatibility, and the validity of nested object structures. If validation fails, the Event-Bus rejects the message before it reaches the business logic layer, preventing malformed or inconsistent data from propagating through the system and ensuring that only well-formed events are processed.

The validation step is implemented inside:

```
TypeAdapter(JobRunnerEvent).validate_python(raw)
```

This mechanism eliminates many runtime failures previously caused by malformed payloads.

### 5.5.2 Failure Scenarios

Several failure scenarios may occur during asynchronous processing.

One possible issue concerns messages missing the `event_name` field.

The Event-Bus explicitly handles this case:

```
if not message_body.get("event_name"):
    logger.warning(...)
    return
```

Another possible failure concerns schema validation errors caused by:

- invalid payload structures,

- missing attributes,
- incompatible data types,
- malformed nested objects.

These failures are intercepted through exception handling:

```
try:
    event: BaseEvent = parse_event(message_body)
except Exception:
    logger.exception("Failed to parse event")
    return
```

The implementation also handles missing consumers:

```
if not handler:
    logger.warning("No handler registered")
    return
```

This prevents unexpected crashes caused by unsupported event types.

Finally, failures occurring inside business logic handlers are managed through the retry mechanisms already provided by Amazon SQS. If processing fails, the message is not acknowledged by the consumer, allowing it to remain in the queue. After the visibility timeout expires, the message automatically becomes available again for reprocessing by another worker instance or by the same consumer. This mechanism ensures reliable delivery and provides a built-in fault-tolerance strategy for transient execution errors.

This mechanism guarantees fault tolerance and resilience against transient failures.

Overall, the introduction of typed validation and centralized event handling significantly improves the reliability and robustness of the asynchronous processing infrastructure.

# Chapter 6

# Evaluation

## 6.1 Qualitative and Quantitative Evaluation

The evaluation of the proposed Typed Event-Bus Registry architecture is conducted using a qualitative analysis of the system improvements introduced over the previous implementation, supported by a discussion of their architectural impact.

A fully quantitative evaluation is limited by the nature of the system, as the benefits introduced by the proposed solution primarily concern structural properties such as maintainability, extensibility, and reliability rather than measurable runtime performance improvements.

For this reason, the evaluation focuses on design-level metrics and architectural characteristics, analyzing how the proposed solution addresses the limitations identified in the previous system.

### 6.1.1 Type Safety and Validation Improvements

A primary improvement introduced by the proposed architecture is the enforcement of strict type safety through typed event schemas.

Compared to the previous implementation, where messages were represented as unstructured JSON objects, the new system introduces explicit validation at the boundary of the processing pipeline. This ensures that only well-formed events are allowed to propagate into the business logic layer.

From a qualitative perspective, this reduces the probability of runtime errors and improves system predictability. It also introduces earlier failure detection, shifting error handling from runtime execution to deserialization time.

Although not directly measurable in terms of performance, this improvement has a significant impact on system robustness and debugging efficiency.

### 6.1.2 Maintainability and Code Organization

Maintainability is one of the most significantly improved aspects of the proposed architecture.

The previous system relied on a centralized dispatcher inside the worker component, which required manual modification for every new event type. This led to increasing complexity and reduced clarity as the system evolved.

In contrast, the Event-Bus Registry introduces a modular structure where handlers are independently defined and registered. This eliminates the need to modify core infrastructure logic when extending the system.

As a result, the codebase becomes more structured, easier to navigate, and more aligned with separation of concerns principles. This improvement is particularly relevant in long-term system evolution scenarios.

### 6.1.3 Extensibility of the Architecture

The proposed solution significantly improves extensibility by decoupling event definitions from routing logic.

New event types can be introduced by simply defining a new schema and registering the corresponding handler, without requiring modifications to existing dispatching mechanisms.

This reduces the risk of introducing regressions in the core worker logic and supports incremental system evolution. From an architectural perspective, this aligns with the open-closed principle, where the system is open to extension but closed to modification.

### 6.1.4 Reliability and Failure Handling

Reliability is improved through early validation and strict schema enforcement.

In the previous architecture, malformed events could propagate deep into the system before being detected, often resulting in runtime exceptions inside business logic components.

The proposed architecture mitigates this issue by validating events at the deserialization stage. Invalid messages are rejected before reaching the processing layer, reducing the likelihood of cascading failures.

Furthermore, the separation between routing logic and business execution reduces the risk that errors in infrastructure components affect unrelated workflows.

## 6.2 Trade-offs and Limitations

### 6.2.1 Increased Architectural Complexity

While the proposed solution improves modularity and maintainability, it introduces additional architectural complexity.

The system now includes multiple abstraction layers, such as typed event schemas, a registry layer, and dynamic handler resolution mechanisms. This increases the conceptual overhead required to understand the full execution flow.

For other small-scale systems, not like our Legenda system, this level of abstraction may not be necessary and could represent an over-engineering of the solution.

### 6.2.2 Development Overhead

Another limitation is the increased initial development effort required to define and maintain typed event schemas.

Each event must be explicitly modeled, validated, and integrated into the registry. This reduces flexibility in rapidly changing environments where event structures evolve frequently.

However, this cost is typically offset by long-term benefits in maintainability and reliability.

### 6.2.3 Runtime Overhead

The introduction of schema validation and dynamic dispatching introduces a minimal runtime overhead during message processing.

Although this overhead is generally negligible compared to network and I/O operations, it is still present and should be considered in high-throughput systems with strict latency requirements.

### 6.2.4 Dependence on External Infrastructure

The proposed architecture continues to rely on Amazon SQS as the underlying messaging system.

This means that inherent limitations of SQS, such as eventual consistency and at-least-once delivery semantics, remain unchanged. Consequently, the system still requires idempotent handlers and careful handling of duplicate messages.

## 6.3 Summary of Evaluation

Overall, the evaluation shows that the proposed Typed Event-Bus Registry architecture introduces substantial improvements in terms of type safety, maintainability, extensibility, and reliability.

These benefits are achieved at the cost of increased architectural complexity and additional development overhead. However, in the context of large-scale distributed systems, these trade-offs are justified by the long-term advantages in system robustness and evolution capability.

# Chapter 7

# Discussion

## 7.1 Generalization of the Approach

The proposed Typed Event-Bus Registry architecture is not strictly tied to the specific implementation described in this work, but can be generalized as a reusable design pattern for event-driven distributed systems.

At its core, the solution introduces a structured layer that formalizes event definition, validation, and dispatching through strongly typed schemas and a centralized registry. This abstraction can be applied to any system that relies on asynchronous message-based communication, independently of the underlying message broker technology.

In particular, the combination of typed events and dynamic handler registration can be adopted in different contexts where systems need to evolve rapidly while maintaining strong guarantees on message structure and processing consistency. This makes the approach suitable for microservices architectures, workflow engines, and event-driven platforms that require a balance between flexibility and reliability.

The general principle underlying this design is the separation between infrastructure-level messaging and application-level event semantics. While tools such as Amazon SQS provide the transport layer, the Event-Bus Registry operates at a higher level of abstraction, ensuring that business events are consistently modeled and safely processed.

## 7.2 Comparison with Alternative Patterns

Several alternative architectural patterns exist for managing event-driven communication, each with different trade-offs in terms of complexity, scala-

bility, and operational overhead.

A first relevant alternative is the use of centralized message dispatchers, similar to the original implementation analyzed in this work. In this approach, event routing is handled through conditional logic structures inside a single consumer component. While this solution is simple to implement, it suffers from poor scalability in terms of maintainability, as the dispatcher becomes increasingly complex with the growth of supported event types.

Another common approach is the adoption of full-fledged event streaming platforms such as Apache Kafka. Kafka introduces a distributed log-based architecture where events are persisted and consumed by multiple independent consumers. Compared to the proposed solution, Kafka provides stronger guarantees in terms of event replayability and throughput scalability. However, it also introduces significantly higher operational complexity and requires additional infrastructure management.

Framework-based event systems, such as those provided by enterprise service buses (ESB), represent another alternative. These systems offer rich routing, transformation, and orchestration capabilities. However, they often introduce tight coupling to the framework itself and may reduce flexibility due to their opinionated nature.

In comparison, the proposed Event-Bus Registry occupies a middle ground between lightweight dispatcher-based systems and heavy distributed streaming platforms. It retains simplicity by leveraging existing infrastructure such as Amazon SQS, while improving structure and maintainability through typed events and dynamic handler registration.

This balance makes the solution particularly suitable for systems that require gradual evolution without introducing excessive infrastructural overhead.

## 7.3 Future Work

Several possible directions for future improvements can be identified based on the current implementation of the architecture.

A first area of enhancement concerns the introduction of advanced observability mechanisms. Although the current design improves traceability through structured events, additional support for distributed tracing systems such as OpenTelemetry could further enhance visibility across service boundaries.

Another potential improvement is the introduction of event versioning mechanisms. As systems evolve, event schemas inevitably change, and sup-

porting backward compatibility at the schema level would further increase robustness in long-running production environments.

A further evolution of the architecture could involve the integration of a dedicated schema registry service. This would centralize the management of event definitions and ensure stronger governance over schema evolution across multiple services.

Finally, performance optimizations and benchmarking could be introduced to evaluate the overhead introduced by validation and dynamic dispatching under high-throughput workloads. This would provide a more quantitative assessment of the trade-offs introduced by the proposed solution.

Overall, these future enhancements would further strengthen the architecture and extend its applicability to larger and more complex distributed systems.

# Chapter 8

# Conclusion

This thesis presented the design and implementation of a Typed Event-Bus Registry architecture aimed at improving the robustness, maintainability, and type safety of asynchronous event-driven systems.

The work was motivated by the limitations observed in a real-world distributed system based on Amazon SQS, where message handling relied on untyped JSON payloads and a centralized dispatcher responsible for routing events through extensive conditional logic. This approach, while functional, introduced significant challenges in terms of scalability, maintainability, and reliability.

To address these issues, a new architecture was proposed based on three main principles: strongly typed event definitions, centralized event registration, and dynamic handler resolution. The introduction of Pydantic-based event schemas enabled strict validation of messages at the boundary of the system, reducing the risk of runtime errors caused by malformed or inconsistent payloads. At the same time, the Event-Bus Registry decoupled event routing from the worker implementation, allowing handlers to be defined and extended independently.

The evaluation of the proposed solution demonstrated clear improvements in key architectural aspects. Type safety was significantly enhanced through explicit schema validation, maintainability was improved by removing centralized routing logic, and extensibility was achieved by enabling modular registration of new event handlers. Furthermore, reliability benefited from earlier detection of invalid messages and a clearer separation of concerns between infrastructure and business logic.

Although the proposed solution introduces additional abstraction layers and a certain degree of development overhead, these trade-offs are justified

in the context of complex and evolving distributed systems. The benefits in terms of system clarity, consistency, and long-term maintainability outweigh the added complexity.

In conclusion, the Typed Event-Bus Registry represents a practical and extensible approach for improving event-driven architectures built on top of message queuing systems. It demonstrates how introducing structured event modeling and dynamic dispatching mechanisms can significantly enhance the quality of distributed systems without requiring changes to the underlying messaging infrastructure.

Future work may further extend this architecture by improving observability, introducing schema versioning mechanisms, and evaluating performance under large-scale production workloads.

# Bibliography


[1] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[2] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2017.

[3] Brendan Burns. *Designing Distributed Systems*. O'Reilly Media, 2016.

[4] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* (2002).

[5] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2003.

[6] Pat Helland. "Life beyond distributed transactions". In: *Communications of the ACM* (2016).

[7] Patrick Eugster et al. "The many faces of publish/subscribe". In: *ACM Computing Surveys* (2003).

[8] Hans-Joachim Happel. "Event-driven architectures". In: (2019).

[9] Amazon Web Services. *Amazon SQS Developer Guide*. 2024. URL: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html.

[10] Monique Snoeck, Stephan Poelmans, and Guido Dedene. "A Layered Software Specification Architecture". In: *Proceedings of the 19th International Conference on Conceptual Modeling*. Vol. 1920. Lecture Notes in Computer Science. Springer, 2000, pp. 454–469. DOI: 10.1007/3-540-45393-8_33.

[11] Peri Tarr et al. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In: *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. IEEE, 1999, pp. 107–119. DOI: 10.1145/302405.302457.

[12] Siva Manikanta Venkatesh Nalam. "Event-Driven Architecture: The Backbone of Real-Time Enterprise Integration". In: *International Journal of Computational and Experimental Science and Engineering* 11.4 (2025). DOI: 10.22399/ijcesen.3983.

[13] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley, 2013.

# Dedication

Quando si raggiunge un traguardo spesso ci si guarda indietro o avanti, io credo che sia sempre più giusto guardarsi intorno, perché è intorno a noi che ci sono le persone che hanno accompagnato il nostro cammino fino a quel punto.

Dedico per questo questa laurea alle persone a me più care, quelle che ci sono state nei momenti più difficili e più bui. Le persone che mi hanno saputo supportare, comprendere ed amare.

Ringrazio per questo mio padre, sempre capace di usare le parole giuste, la comprensione e l'empatia necessarie, riuscendo a tramandarmi tanto amore e supporto.

Mia madre, capace sempre di farmi trovare la forza per continuare a combattere, rialzarmi dalle mie sconfitte, dalle mie ansie e dalle mie paure, ricordandomi spesso e sempre di dover essere anche delicato con me stesso.

I mie fratelli, Davide ed Enrico, compagni, amici e fratelli per me. Sono orgoglioso di entrambi e delle persone che sono, so di poter contare sempre su di loro e loro potranno contare sempre su di me.

Ai miei amici, che anche a distanza di kilometri hanno saputo esserci, anche con un lungo o breve audio, ringrazio la mia gang del liceo: Alessandro, Mariano, Vincenzo e Giovanni; e ringrazio le nuove persone che sono entrate a far parte della mia vita come Massimo.

Dedico poi questo traguardo anche a quella persona che occupa un posto nel mio cuore. Cara Elena, non so cosa il futuro porta con sé, ma so quello che sei stata per me in questi anni e quindi ringrazio anche te per essermi stata vicino come hai potuto.

Il futuro porta con sé tante scelte, strade, ostacoli e difficoltà, non c'è nulla che finisce realmente. Ogni conclusione non è mai un finale, ma solo un nuovo inizio.