BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION COMPUTER SCIENCE IN ENGLISH

DIPLOMA THESIS

An Empirical Validation of Scalability and Fault Tolerance Patterns in a.NET Microservices Architecture on Kubernetes

Supervisor
Assoc. Prof. Rares Florin Boian Ph.D.

Author
Cujba Daniel

2025

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICA IN LIMBA ENGLEZA

LUCRARE DE LICENŢĂ

O validare empirică a modelelor de scalabilitate și toleranță la erori într-o arhitectură de microservicii .NET pe Kubernetes

Conducător științific
Conf. Dr. Boian Rares

Absolvent
Cujba Daniel

2025

ABSTRACT

The imperative for modern software systems to be both highly scalable and resilient to failure has driven the widespread adoption of microservice architectures. However, the theoretical benefits of this architectural style are not inherently guaranteed and depend on the deliberate application and validation of specific design patterns and technologies. This thesis addresses the gap between theoretical principles and practical implementation by presenting the design, end-to-end development, and empirical validation of a cloud-native e-commerce application. The system is architected using.NET microservices, containerized and orchestrated with Kubernetes, and employs RabbitMQ for asynchronous, decoupled communication. Application-level fault tolerance is implemented through the Polly library, integrating Retry and Circuit Breaker patterns to enhance inter-service communication robustness. To validate the architecture, a sustained 30-minute load test was conducted on a key service, demonstrating high performance with a stable throughput of approximately 124 requests per second at a median latency of 35ms, with zero failures. Crucially, the results confirmed that the service becomes CPU-bound under load, a behavior that correctly triggers the configured Kubernetes Horizontal Pod Autoscaler (HPA) policy. This work's primary contribution is an empirically validated architectural blueprint, demonstrating the effective integration of specific technologies and patterns to build verifiably scalable and fault-tolerant.NET microservice applications.

DISCLAIMER

During the preparation of this work the author used Google Gemini and DeepSeek in order to improve wording, rephrase sentences, check for spelling, and assist with the generation of LaTeX code. After using these tools/services, the author reviewed and edited the content as needed and takes full responsibility for the content of the thesis.

Contents

Chapter 1

Introduction

1.1 Background: The Ascendancy of Microservices in Distributed System Design

The landscape of software architecture has undergone a significant transformation over the past decade, marked by a discernible shift from monolithic application designs towards microservice architectures. This paradigm shift is driven by the escalating demands of modern digital enterprises for increased agility, faster development cycles, independent deployability of components, and the flexibility to utilize diverse technology stacks for different functionalities. Monolithic systems, characterized by a single, large codebase where all functionalities are tightly coupled, often struggle to meet these demands, particularly in terms of scaling specific parts of an application or adopting new technologies without impacting the entire system. Microservices address these limitations by decomposing applications into a collection of small, autonomous services, each responsible for a specific business capability. [FL14] [Ric18] [Eva03] This architectural style inherently leads to distributed systems, as services typically run in separate processes and communicate over a network.

The adoption of microservices is not merely a technical trend but a strategic response to the need for systems that can evolve rapidly and scale efficiently. Enterprises such as Netflix and Amazon have famously transitioned to microservices to enhance their scalability, foster innovation, and ensure long-term growth. [She24] [Wal25] However, the very nature of microservices introduces a new set of challenges. While they offer solutions to the rigidity of monoliths, their distributed characteristics bring forth complexities in managing inter-service communication, ensuring data consistency across disparate services, and maintaining overall system reliability. The independent operation of services, a key benefit, also means that the system as a whole must be resilient to failures in its individual parts. Thus,

the pursuit of scalability and fault tolerance becomes a central concern in microservice design, moving beyond the capabilities offered by monolithic structures but demanding new architectural considerations.

1.2 Problem Statement: The Intrinsic Scalability and Fault Tolerance Imperatives in Microservice Ecosystems

While microservice architectures present a compelling proposition for building scalable and resilient applications, these benefits are not inherently guaranteed. The distributed and interdependent nature of microservices makes them susceptible to unique failure modes and performance bottlenecks that differ significantly from those encountered in monolithic systems. Without deliberate and sophisticated design and engineering practices, the potential advantages of microservices can be undermined by the complexities they introduce. Key challenges include the risk of cascading failures, where the failure of one service can propagate and impact dependent services, leading to widespread outages. Network latency, an unavoidable aspect of inter-service communication, can degrade performance and must be carefully managed. Ensuring data consistency across multiple, independently managed databases associated with different services is another significant hurdle. Furthermore, the operational overhead of deploying, managing, and monitoring a multitude of small services can be substantial.

The amplification effect of failures is a particularly critical concern. In a tightly interconnected microservice ecosystem, a seemingly minor, localized issue within a single service—such as a transient network glitch or a bug causing slow responses—can have a disproportionately large impact if not effectively isolated. This is because services often rely on other services to fulfill user requests. [Nyg18] If a downstream service becomes unresponsive or erroneous, upstream services that depend on it may also degrade or fail. This ripple effect underscores the non-negotiable requirement for robust fault tolerance strategies that go beyond simple error handling. The architectural design must inherently support the detection, isolation, and graceful handling of failures to prevent them from escalating into system-wide disruptions.

1.3 Research Objectives and Guiding Questions

This thesis aims to conduct a comprehensive and critical examination of the principles, patterns, technologies, and practices essential for achieving high levels of

scalability and fault tolerance in microservice-based applications. The specific objectives are:

1. To critically evaluate existing architectural principles and design patterns that underpin scalable and fault-tolerant microservice designs.
2. To analyze the influence of different inter-service communication paradigms (e.g., synchronous RESTful APIs versus asynchronous event-driven messaging) on the system's ability to scale and tolerate faults.
3. To investigate the role of modern containerization, orchestration platforms (particularly Kubernetes)
4. To synthesize these findings into a holistic understanding of how to architect microservices that are both adaptable to varying loads and robust in the face of partial failures.

To achieve these objectives, this research will be guided by the following key questions:

- What are the fundamental architectural principles that underpin scalable and fault-tolerant microservice designs?
- How do different inter-service communication paradigms (e.g., synchronous REST vs. asynchronous messaging) influence scalability and fault tolerance?
- Which design patterns (e.g., Circuit Breaker, Bulkhead, Retry, API Gateway, Load Balancing) are most effective in addressing specific scalability and fault tolerance challenges, and under what conditions?
- What is the role of containerization (Docker) and orchestration platforms (Kubernetes, KEDA) in enabling dynamic scaling and resilient deployments?
- How can chaos engineering be systematically applied to proactively identify and mitigate weaknesses in microservice resilience?
- What are the key considerations for data management and consistency in distributed microservice environments to ensure both scalability and fault tolerance?

1.4 Contributions of the Thesis

This thesis endeavors to make several contributions to the understanding of scalability and fault tolerance in microservice architectures. Firstly, it will provide a

synthesized framework, drawing from academic research and industry best practices, for designing microservice systems that are inherently resilient and scalable. This framework will integrate insights from foundational principles with the application of specific design patterns and enabling technologies. Secondly, the thesis will offer a comparative analysis of different technological approaches and patterns, highlighting their respective strengths, weaknesses, and optimal use contexts. This includes a nuanced discussion of communication styles, data management strategies, and resilience mechanisms. Thirdly, by integrating the concept of proactive resilience through practices like chaos engineering, the thesis aims to shift the discourse from merely reactive fault handling to a more holistic, preventative engineering approach. Finally, it seeks to identify key best practices and illuminate promising future research directions in this rapidly evolving field, leveraging insights from the provided source document and a broader corpus of scholarly and technical literature. The overarching goal is to furnish a comprehensive, academically rigorous perspective that bridges theoretical understanding with practical application.

1.5 Thesis Structure

The remainder of this thesis is organized as follows:

- Chapter 2 lays the theoretical foundations of microservice architecture, focusing on aspects pertinent to building resilient and scalable systems. It defines microservices, discusses their core characteristics, advantages, and disadvantages in distributed contexts, and examines critical architectural considerations such as inter-service communication and data management.
- Chapter 3 delves into design patterns and strategies specifically aimed at achieving scalability in microservices. Topics include horizontal and vertical scaling, load balancing and API gateways.
- Chapter 4 focuses on design patterns and strategies for enhancing fault tolerance. It covers principles like redundancy and isolation, and patterns such as Circuit Breaker, Retry and Bulkhead.
- Chapter 5 outlines the application's microservice architecture, detailing the public e-commerce storefront for customers and the separate, secure backend for administrative inventory management.
- Chapter 6 presents the results of a successful load test which proved the InventoryService is stable and performant, validating that its CPU-bound behavior correctly triggers the designed auto-scaling policy.

- Chapter 7 concludes the thesis with a summary of key findings and contributions, discusses the limitations of the study, and suggests avenues for future research.
- The Bibliography lists all cited works.

Chapter 2

Theoretical Foundations of Microservices for Resilient and Scalable Systems

2.1 Defining Microservice Architecture: A Paradigm for Distributed Functionality

Microservice architecture is an architectural style that structures an application as a collection of small, autonomous, and independently deployable services. [FL14] [Ric18] Each service is designed around a specific business capability and can be developed, deployed, operated, and scaled independently of other services. This contrasts sharply with traditional monolithic architectures, where all application functionalities are interwoven into a single, large, and tightly coupled codebase. In a monolithic system, scaling a specific feature or updating a single component often requires redeploying the entire application, leading to slower release cycles and increased risk.

The independence of microservices is a cornerstone of their design philosophy. This independence extends to technology choices, allowing different services to be implemented using different programming languages, frameworks, and data storage technologies best suited for their specific tasks. Communication between these services typically occurs over a network using lightweight protocols such as HTTP/REST or asynchronous messaging mechanisms. While this independence offers significant advantages in terms of flexibility, team autonomy, and targeted scalability, it also introduces the complexities inherent in distributed systems. The network becomes a critical component, and inter-service communication reliability and performance are paramount. Thus, the very characteristics that enable benefits like independent scaling also necessitate robust mechanisms for managing interac-

tions and potential failures in a distributed environment, forming a central theme in the pursuit of scalability and fault tolerance.

2.2 Core Characteristics Enabling (and Complicating) Scalability and Fault Tolerance

Several core characteristics of microservices directly influence their capacity for scalability and fault tolerance, often presenting both opportunities and challenges.

- Independently Deployable: Each microservice can be deployed and updated without affecting others. This allows for rapid iteration and targeted scaling of individual services based on their specific load, rather than scaling the entire application. However, managing deployments and ensuring compatibility between versions of interacting services becomes a new challenge.
- Loosely Coupled: Services are designed to minimize dependencies on each other, interacting through well-defined APIs. Loose coupling is crucial for fault isolation; a failure in one service is less likely to cascade to others if dependencies are managed carefully. Achieving true loose coupling, however, requires meticulous API design and governance.
- Technology Agnostic: The ability to use different technologies for different services allows teams to choose the best tools for the job. While this fosters innovation, it can also increase operational complexity in terms of monitoring, managing diverse runtime environments, and ensuring consistent security practices.
- Organized around Business Capabilities (Domain-Driven Design): Microservices are often aligned with specific business domains. This promotes a clear separation of concerns and allows teams to develop deep expertise. From a scalability perspective, it means that services critical to high-demand business functions can be scaled independently.

  D
- ecentralized Governance and Data Management: Teams often have autonomy over their services, including their data persistence strategies (e.g., database-per-service). This supports independent evolution and scaling but complicates tasks like ensuring data consistency across services or performing queries that span multiple domains.

The "small service" characteristic, while central to the definition, presents a "granularity dilemma." If services are decomposed too finely (sometimes referred to as "nanoservices"), the volume of inter-service communication can become excessive. [She24] [BIG19] Each network call introduces latency and a potential point of failure. Therefore, determining the appropriate size and scope for each microservice is a critical design decision. It involves balancing the benefits of modularity and independent scaling against the increased communication overhead and the expanded surface area for potential failures that arise from a highly distributed system. An optimal granularity minimizes unnecessary chattiness between services while still allowing for focused functionality and independent evolution.

2.3 Advantages and Disadvantages in Distributed Contexts

The adoption of microservices offers a compelling set of advantages, particularly for applications requiring high scalability and resilience, but these are accompanied by inherent challenges stemming from their distributed nature.

Advantages:

- Improved Scalability: Individual services can be scaled independently based on their specific resource needs and demand, leading to more efficient resource utilization compared to scaling an entire monolith.
- Faster Time-to-Market: Smaller, focused teams can develop, test, and deploy services independently, accelerating release cycles.
- Enhanced Flexibility and Technology Diversity: Teams can choose the most appropriate technology stack for each service, fostering innovation and allowing for easier adoption of new technologies.
- Better Fault Isolation: A failure in one microservice, if properly handled, is less likely to bring down the entire application. Other services can continue to operate, improving overall system resilience. [Wal25] [Mak23]
- Increased Developer Productivity: Smaller codebases are easier to understand, maintain, and test. Teams can work in parallel with fewer dependencies.

Disadvantages (Challenges):

- Complexity: Managing a distributed system of many small services is inherently more complex than managing a single monolith. This includes aspects of deployment, monitoring, inter-service communication, and debugging.

- Data Management: Ensuring data consistency and integrity across multiple services, each potentially with its own database, is a significant challenge. Distributed transactions are complex and often avoided in favor of eventual consistency models, which require careful application design.
- Network Latency and Reliability: Communication between services occurs over a network, introducing latency and the possibility of network failures. These must be accounted for in performance and reliability design.
- onitoring and Debugging: Tracing requests and diagnosing issues across multiple services can be significantly more difficult than in a monolithic application, necessitating sophisticated observability tools.
- Deployment and Orchestration: Managing the deployment, scaling, and lifecycle of numerous microservices requires robust automation and orchestration tools.
- Security: Securing inter-service communication and managing identities and access control across a distributed system presents additional complexities.

The following table provides a comparative overview of microservices and monolithic architectures, highlighting key differences relevant to scalability and fault tolerance.

Feature	**Microservices**	**Monolithic Architecture**
Structure	Distributed, loosely coupled	Single, tightly integrated
Scalability	Independent, granular	Limited, coarse-grained
Technology	Diverse, technology agnostic	Homogeneous, technology dependent
Fault Isolation	High, localized failures	Low, system-wide failures
Development	Parallel, cross-functional teams	Sequential, siloed teams
Maintenance	Easier, focused services	Difficult, monolithic codebase
Time-to-Market	Faster, agile practices	Slower, waterfall practices
Complexity	Higher, distributed systems	Lower, single codebase
Data Management	Challenging, distributed data	Easier, centralized data

Table 2.1: Comparative Overview of Microservices and Monolithic Architectures [Ric18]

Chapter 3

Design Patterns and Strategies for Scalability in Microservices

Scalability refers to a system's ability to handle an increasing amount of work by adding resources, or its ability to handle a decreasing amount of work by removing resources. In the context of microservices, two primary dimensions of scaling are considered: vertical scaling and horizontal scaling.

3.1 Vertical Scaling

Vertical scaling (scaling up/down) involves increasing or decreasing the resources (e.g., CPU, RAM, storage) of an existing service instance. For example, moving a service to a more powerful server or allocating more memory to its container. While conceptually simple, vertical scaling has inherent limitations. There's an upper bound to the resources a single instance can effectively utilize, and it often involves downtime or service interruption to apply changes. Moreover, it doesn't inherently improve fault tolerance through redundancy.

3.2 Horizontal Scaling

Horizontal scaling (scaling out/in) involves adding more instances of a service or removing existing instances to distribute the load. Microservice architectures are particularly well-suited for horizontal scaling due to their design principles of small, independent, and often stateless services. Each instance can handle a subset of the incoming requests, and load balancers distribute traffic among them. Horizontal scaling is the predominant strategy for cloud-native microservices. This preference stems from its superior elasticity, as instances can be dynamically added or removed based on real-time demand, often facilitated by orchestration platforms like Kuber-

netes. [Kub25] [BGO+16] It also inherently provides a degree of fault tolerance: if one instance fails, others can continue to serve traffic. Furthermore, horizontal scaling can be more cost-effective at scale, utilizing commodity hardware or smaller virtual machine instances rather than relying on expensive, high-end servers required for significant vertical scaling. The ability to independently scale individual microservices based on their specific load profiles is a key advantage over monolithic systems, where the entire application must be scaled even if only one component is a bottleneck.

3.3 Load Balancing Techniques and the Pivotal Role of API Gateways

When microservices are scaled horizontally, load balancing becomes essential to distribute incoming requests effectively across the multiple available instances of a service. The goal is to ensure even resource utilization, prevent any single instance from becoming a bottleneck, and improve overall application responsiveness and availability. Various load balancing algorithms can be employed, including:

- Round Robin: Distributes requests sequentially to each server in a list.
- Least Connections: Directs requests to the server with the fewest active connections.
- Weighted Round Robin/Least Connections: Assigns weights to servers, allowing more powerful servers to receive a proportionally larger share of the traffic.
- IP Hash: Directs requests from a specific client IP address to the same server, which can be useful for maintaining session affinity if required (though stateless services are preferred).

API Gateways play a pivotal role in modern microservice architectures, often acting as the primary entry point for all external client requests. Beyond simple request routing, an API Gateway can perform sophisticated load balancing across downstream microservice instances. It abstracts the complexity of the internal microservice landscape from clients, providing a single, stable interface.

The API Gateway is more than just a traffic director; it serves as a critical control plane for enhancing both scalability and resilience. By centralizing access, it can implement various policies that benefit the entire system:

- Request Routing and Composition: Directing requests to appropriate service versions or aggregating results from multiple services.

- Offloading Cross-Cutting Concerns: Handling tasks like SSL termination, authentication, authorization, and logging, freeing individual microservices from these responsibilities.
- Caching: Caching responses from frequently accessed, relatively static backend services at the gateway level can significantly reduce the load on these services and improve response times for clients.
- Rate Limiting and Throttling: Protecting backend services from being overwhelmed by excessive requests, whether due to legitimate traffic spikes or malicious attacks.
- Protocol Translation: Allowing clients using one protocol (e.g., HTTP/1.1) to communicate with services using another (e.g., gRPC).

3.4 Asynchronous Processing and Message Queues for Decoupled Scaling

Asynchronous communication, facilitated by message queues, is a powerful paradigm for building scalable and resilient microservice architectures. By decoupling services, message queues allow producers (services that send messages) and consumers (services that process messages) to operate and scale independently, effectively handling variable loads and transient failures. [Rab25]

Message queues like Apache Kafka and RabbitMQ act as intermediaries or buffers between services. When a producer service needs to initiate a task or notify another service of an event, it sends a message to a queue instead of making a direct synchronous call. The message is persisted in the queue, and the producer can continue its operations without waiting for the message to be processed. Consumer services subscribe to the queue and pull messages for processing when they have available capacity.

This decoupling provides several scalability benefits:

- Load Leveling: Message queues can absorb sudden spikes in requests (messages). If producers generate messages at a higher rate than consumers can process, the messages accumulate in the queue. Consumers can then process this backlog at their own sustainable pace, preventing them from being overwhelmed. This smooths out load variations and improves system stability.
- Independent Scaling: Producer and consumer services can be scaled independently based on their respective workloads. If the message queue depth grows

consistently, it indicates that consumers are a bottleneck, and more consumer instances can be added. Conversely, if producers are generating a high volume of messages, they can be scaled out without immediately impacting consumers.

- Improved Responsiveness: Producer services are not blocked waiting for downstream processing, leading to lower latency and better responsiveness for operations initiated by producers.
- Enhanced Resilience: If a consumer service fails or is temporarily unavailable, messages remain in the queue and can be processed once the service recovers. This prevents data loss and allows the system to gracefully handle transient outages of consumer services.

Chapter 4

Design Patterns and Strategies for Fault Tolerance in Microservices

Fault tolerance is the ability of a system to continue operating, potentially at a reduced level, rather than failing completely when one or more of its components fail. In microservice architectures, where applications are composed of numerous interdependent services, designing for fault tolerance is paramount. Three core principles underpin most fault tolerance strategies:

1. **Redundancy:** This involves deploying multiple instances of critical components (e.g., microservices, databases, load balancers) so that if one instance fails, others can take over its workload. Horizontal scaling, as discussed for scalability, inherently provides redundancy. Replicating data across multiple servers or data centers is another form of redundancy crucial for data availability and durability.

2. **Isolation:** This principle aims to prevent failures in one part of the system from cascading and affecting other parts. Patterns like Bulkhead (discussed later) are designed to contain the impact of a failing service by isolating the resources it consumes. Decomposing an application into microservices itself is a form of isolation, as a failure in one service ideally should not bring down unrelated services.

3. **Graceful Degradation:** When a complete failure of a component occurs and redundancy or immediate recovery is not possible, the system should aim to degrade gracefully rather than failing catastrophically. This means providing partial functionality or a reduced level of service to users, maintaining essential operations while non-critical features might be temporarily unavailable. Fallback mechanisms are key to achieving graceful degradation.

Modern approaches to fault tolerance in microservices emphasize proactive design

rather than solely reactive measures. This involves anticipating potential failure modes during the design phase and building in mechanisms to detect, isolate, and handle these failures automatically. Patterns like Circuit Breakers, Retries, and Timeouts are not afterthoughts but integral parts of a resilient service's design. Furthermore, advanced practices like chaos engineering take this proactivity a step further by deliberately injecting failures into test and even production environments to uncover weaknesses and validate the effectiveness of fault tolerance strategies before they are triggered by real-world incidents. This signifies a maturation in understanding that failures in distributed systems are inevitable and must be systematically planned for from the outset.

4.1 The Circuit Breaker Pattern: Preventing Cascading Failures

The Circuit Breaker pattern is a critical fault tolerance mechanism designed to prevent an application from repeatedly trying to execute an operation that is likely to fail, thereby protecting both the calling service from resource exhaustion and the failing service from being overwhelmed by repeated requests. [Nyg18] [Pol25] [Mak23] [IA22] It acts like an electrical circuit breaker: if a downstream service starts to exhibit a high failure rate, the circuit breaker "trips" or "opens," and subsequent calls from the client are immediately failed without attempting to contact the problematic service. This prevents cascading failures where one failing service causes dependent services to also fail or become unresponsive.

A circuit breaker typically operates in three states :

1. **Closed:** In the normal state, requests are allowed to pass through to the downstream service. The circuit breaker monitors the number of failures (e.g., timeouts, exceptions). If the failure count exceeds a configured threshold within a specific time window, the breaker transitions to the Open state.

2. **Open:** When the circuit is open, requests to the downstream service are immediately rejected (e.g., by returning an error or a predefined fallback response) without making an actual call. This gives the failing service time to recover. After a configured timeout period, the breaker transitions to the Half-Open state.

3. **Half-Open:** In this state, a limited number of test requests are allowed to pass through to the downstream service. If these requests succeed, the breaker assumes the service has recovered and transitions back to the Closed state (re-

setting the failure counter). If any of these test requests fail, the breaker reverts to the Open state, and the recovery timeout period begins again.

4.2 Retry Mechanisms

Retry mechanisms are designed to handle transient faults—temporary issues like network glitches, brief service unavailability, or momentary resource contention—by automatically re-attempting a failed operation. The assumption is that the fault is short-lived and a subsequent attempt is likely to succeed. Retries can significantly improve the resilience of inter-service communication.

However, naive retry implementations can cause more harm than good. If a service is struggling due to overload, immediate and repeated retries from multiple clients can exacerbate the problem, leading to a ”retry storm” that prevents the service from recovering. To mitigate this, effective retry strategies must incorporate:

- **Backoff Policies:** Instead of retrying immediately, a delay is introduced between attempts. Common backoff strategies include:
  - **Exponential Backoff:** The delay between retries increases exponentially (e.g., 1s, 2s, 4s, 8s). This gives the failing service progressively more time to recover. [Pol25]
  - **Jitter:** Adding a small random amount of time to the backoff delay helps to prevent synchronized retries from multiple clients, which could still overwhelm the service. [Pol25]
- **Maximum Retry Attempts:** Limiting the number of retries prevents indefinite attempts for persistent failures.
- **Retryable Conditions:** Not all errors should be retried. For example, a ”400 Bad Request” error is unlikely to succeed on retry without modification, whereas a ”503 Service Unavailable” might.

4.3 The Bulkhead Pattern: Isolating Resources for Fault Containment

The Bulkhead pattern is a fault isolation technique inspired by the watertight compartments (bulkheads) in a ship’s hull, which prevent a single breach from flooding the entire vessel. [She24] [BIG19] In microservice architectures, this pattern aims to isolate resources used for communicating with different downstream services or

handling different types of requests. If one downstream service becomes slow, unresponsive, or fails, the resources allocated for interacting with it (e.g., connection pools, thread pools) may become exhausted. Without bulkheads, this exhaustion could impact the calling service's ability to interact with other, healthy downstream services, leading to a cascading failure.

By partitioning resources, the Bulkhead pattern ensures that a failure or slowdown in one area is contained and does not deplete resources needed by other parts of the application. For example, a microservice might maintain separate thread pools for calls to Service A and Service B. If Service A becomes unresponsive and ties up all threads in its dedicated pool, calls to Service B can still proceed using their separate thread pool.

The concept of bulkheads can be applied more broadly than just connection and thread pools. It can extend to:

- **Isolating Message Queues:** If different types of critical tasks or events are processed via message queues, using separate queues for distinct functionalities can act as a bulkhead. If one type of task floods its queue or its consumers fail, it won't impede the processing of messages from other queues.
- **Infrastructure Segmentation:** Deploying different groups of critical services on separate Kubernetes clusters, distinct sets of virtual machines, or in different availability zones can be viewed as a higher-level application of the bulkhead principle. This helps to contain the "blast radius" of infrastructure-level failures.
- **Process-Level Isolation:** Running different services in separate processes or containers is a fundamental form of bulkhead, ensuring that a crash in one service doesn't directly bring down others.

Chapter 5

Building applications with Microservices Architecture in .NET

This chapter delves into the critical aspects of scalability and fault tolerance in the context of microservice-based applications. Using a practical example of an e-commerce platform built with .NET, React, and deployed on Kubernetes, we explore the architectural decisions and implementation details that contribute to a resilient and scalable system. We analyze the role of various technologies and patterns, including API Gateways, asynchronous messaging with RabbitMQ, and the application of resilience policies using Polly. The discussion highlights how these components work in concert to handle fluctuating loads and gracefully manage failures, ensuring a seamless user experience.

5.1 Architectural Overview

The application's architecture is a classic example of a microservice-based system. A user-facing React application and a separate React Admin interface for administrative tasks serve as the clients. All client requests are first routed to an API Gateway built with YARP (Yet Another Reverse Proxy). This gateway acts as a single, unified entry point, simplifying the frontend logic and directing traffic to the appropriate backend microservice.

The core backend logic is distributed across three distinct services:

- OrderService: Manages the creation and tracking of customer orders.
- InventoryService: Handles product stock levels and availability.
- PaymentService: Integrates with Stripe to process payments securely.

Communication between these services is handled through two distinct channels. For immediate, request-response interactions, services communicate synchronously

via direct HTTP calls. For processes that can be handled in the background, such as updating inventory after a payment, the services communicate asynchronously using RabbitMQ as a message broker. This dual approach optimizes for both responsiveness and resilience.

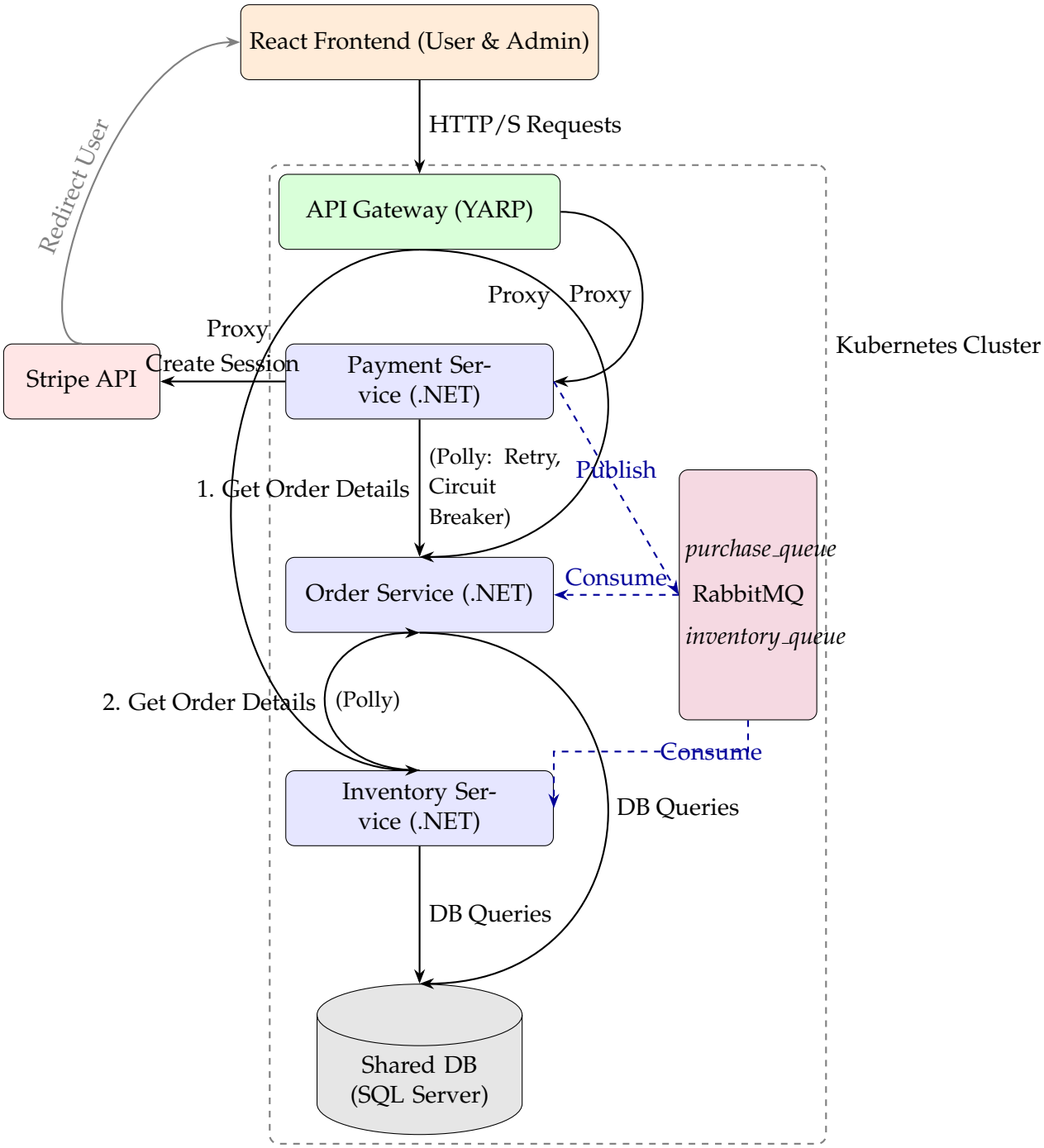

Figure 5.1: The microservice architecture for the e-commerce application. It demonstrates a vertical flow from the API Gateway to the services, which use a shared database. Asynchronous communication via RabbitMQ and resilient inter-service calls with Polly are key features for fault tolerance.

5.2 Implementing Scalability

Scalability is addressed at multiple levels within this architecture, from the infrastructure to the application logic itself.

5.2.1 Service-Level Scalability with Kubernetes

The entire backend is designed for a cloud-native environment, with each microservice containerized and deployed on Kubernetes. This platform provides a powerful framework for achieving automated, horizontal scalability.

The deployment of each service is managed by three key Kubernetes objects. First, a Deployment manifest defines the desired state of the service. It specifies the container image to use, resource requests and limits to ensure predictable performance, and any necessary environment configurations.

Second, a Service manifest exposes the pods managed by the Deployment to the network. It provides a stable IP address and DNS name, and it automatically load-balances incoming traffic across all available replicas of the microservice. In this application, the InventoryService/OrderService/PaymentService is exposed via a LoadBalancer service, making it accessible from outside the cluster on a specific port.

Finally, a HorizontalPodAutoscaler (HPA) automates the scaling process. The InventoryService/OrderService/PaymentService is configured with an HPA that monitors its CPU utilization. The system is set to maintain a minimum of one running instance but can automatically scale up to a maximum of three instances.[Kub25] [BIG19] A scale-out event is triggered whenever the average CPU utilization across all pods exceeds 80%, ensuring that the application can seamlessly handle sudden spikes in traffic without manual intervention. (See Figure 5.2 for a visual representation of the HPA mechanism.)

5.2.2 Asynchronous Communication with RabbitMQ

The use of RabbitMQ for inter-service communication is a cornerstone of the application's scalability strategy. [Rab25] When an order is placed and payment is completed, the system does not rely on a chain of direct, blocking calls. Instead, the PaymentService publishes messages to designated queues in RabbitMQ.

This decouples the services, allowing them to operate and scale independently. The OrderService and InventoryService each contain a background consumer that listens for relevant messages. This asynchronous, event-driven pattern prevents the PaymentService from being blocked while waiting for downstream services to

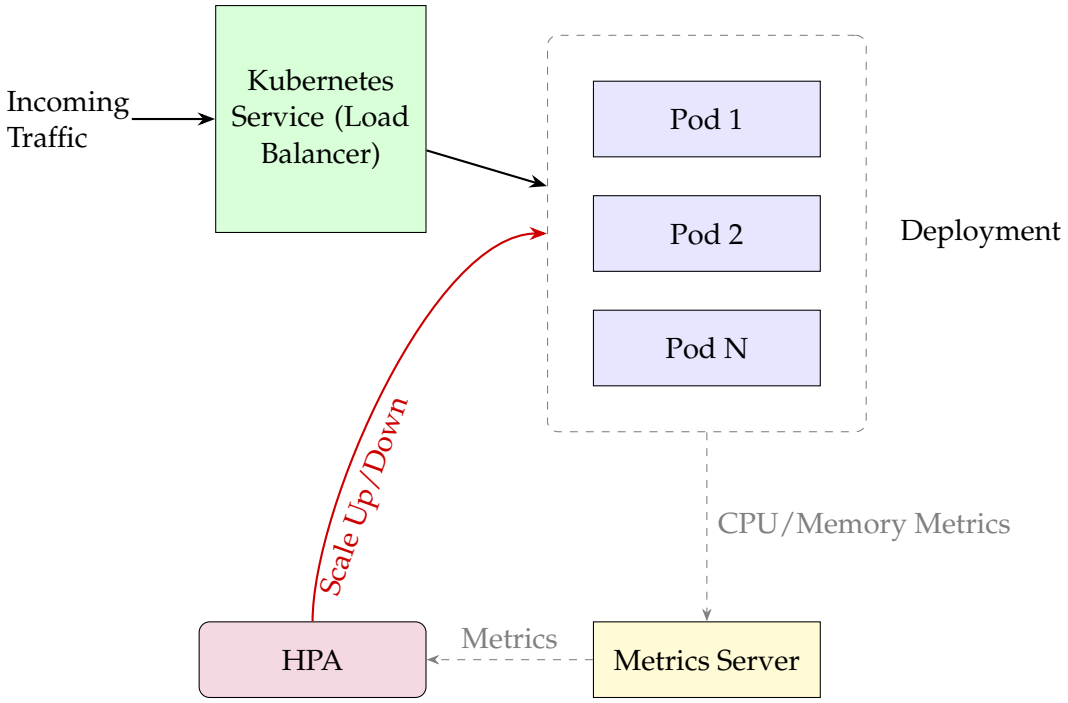

Figure 5.2: A horizontally oriented view of the Kubernetes autoscaling mechanism. Incoming traffic is directed by the Kubernetes Service to the Deployment, which manages a scalable set of Pods. The Horizontal Pod Autoscaler (HPA) forms a control loop below: it receives metrics from the Pods via the Metrics Server and adjusts the number of replicas in the Deployment accordingly.

confirm updates. The result is improved responsiveness for the end-user and higher overall system throughput, as each service can process tasks at its own pace.

5.3 Resilient Communication with Polly

The .NET resilience and transient-fault-handling library, Polly, is integrated into the application's HTTP clients to gracefully handle the inevitable failures in a distributed system.

5.3.1 The Retry Pattern

To combat transient network issues or temporary service unavailability, a Retry policy is implemented. This policy automatically re-attempts a failed HTTP request a specified number of times. To avoid overwhelming a struggling service, it uses an exponential backoff strategy, where the delay between each retry attempt increases progressively. This simple yet effective pattern significantly improves the reliability of inter-service communication. [Pol25]

5.3.2 The Circuit Breaker Pattern

To prevent cascading failures, where a problem in one service brings down others, the Circuit Breaker pattern is employed. This policy monitors calls to a downstream service. If the number of consecutive failures surpasses a configured threshold (configured as 10 failed requests), the circuit "opens," and all subsequent calls fail immediately, without making a network request. This protects the calling service from wasting resources on calls that are likely to fail. After a timeout period, the circuit enters a "half-open" state, allowing a single test request through. If this request succeeds, the circuit "closes," and normal operation resumes. Otherwise, it remains open, giving the failing service more time to recover. [Pol25]

5.3.3 Fault Tolerance with Kubernetes

Kubernetes itself provides a powerful, declarative foundation for fault tolerance. Its control plane continuously works to ensure the actual state of the cluster matches the desired state defined in the configuration files.

- **Self-Healing:** If a container or pod crashes for any reason, Kubernetes automatically detects the failure and restarts it or provisions a new one, ensuring the service remains available without manual intervention.
- **Rolling Updates:** When deploying a new version of a service, Kubernetes performs a rolling update. It gradually replaces old pods with new ones, ensuring there is no downtime. It also monitors the health of the new pods and can automatically roll back to the previous version if they fail to start correctly.

5.3.4 Message Durability with RabbitMQ

For true fault tolerance in a production system, the RabbitMQ queues are configured as durable. This ensures that messages persist on disk and will not be lost even if the RabbitMQ server restarts. The application already implements manual message acknowledgments, meaning a message is only removed from the queue after the consumer service has successfully processed it. This combination of durable queues and acknowledgments guarantees that no critical business event, like an inventory update, is lost due to a transient failure.

5.4 Observability: Metrics and Monitoring

While scalability and fault tolerance mechanisms provide the foundation for a robust system, they are incomplete without a comprehensive observability strategy. In

a distributed microservices architecture, understanding the internal state and performance of the system is paramount for maintaining reliability.

The application employs a modern observability stack consisting of Prometheus for metrics collection and Grafana for visualization. [Pro25] [Gra25]

Each .NET microservice is instrumented using the prometheus-net library, which exposes a standard /metrics endpoint. This endpoint provides a wealth of real-time data, including HTTP request rates, error percentages, and response latencies, as well as .NET runtime statistics like garbage collection and CPU usage.

To collect this data, Prometheus is deployed within the Kubernetes cluster. The collection process is automated using ServiceMonitor resources. For each microservice (InventoryService, OrderService, and PaymentService), a ServiceMonitor tells the Prometheus instance how to discover and "scrape" (or pull) the data from its /metrics endpoint. This declarative approach integrates seamlessly with Kubernetes service discovery, ensuring that as services scale up or down, all new instances are automatically included in the monitoring. [Pro25]

While Prometheus is the engine for collecting and storing this time-series data, Grafana serves as the visualization layer. Operators and developers can build detailed dashboards in Grafana to plot key metrics over time. [Gra25] These dashboards provide at-a-glance visibility into the health of the entire system. For example, one could visualize the CPU utilization of the InventoryService pods to see the Horizontal Pod Autoscaler in action, or monitor the HTTP 5xx error rate across all services to detect failures. This ability to monitor, visualize, and alert on key performance indicators is not just a reactive tool for debugging but a proactive one for identifying performance bottlenecks and potential failures before they impact users.

5.5 Application User Interface and Workflow

This section details the user-facing and administrative interfaces of the e-commerce application, illustrating the typical user journey from browsing products to completing a purchase, as well as the administrative workflow for managing inventory.

5.5.1 Customer-Facing Storefront

The customer experience is designed to be clean, intuitive, and efficient, guiding the user from product discovery to checkout with ease.

- Product Catalog View (Homepage): The main entry point for customers is a modern storefront that displays products in a clear grid layout. Each item is presented as a distinct card containing a product image, name, a brief description, and the price. An "Add to Cart" button is prominently featured on

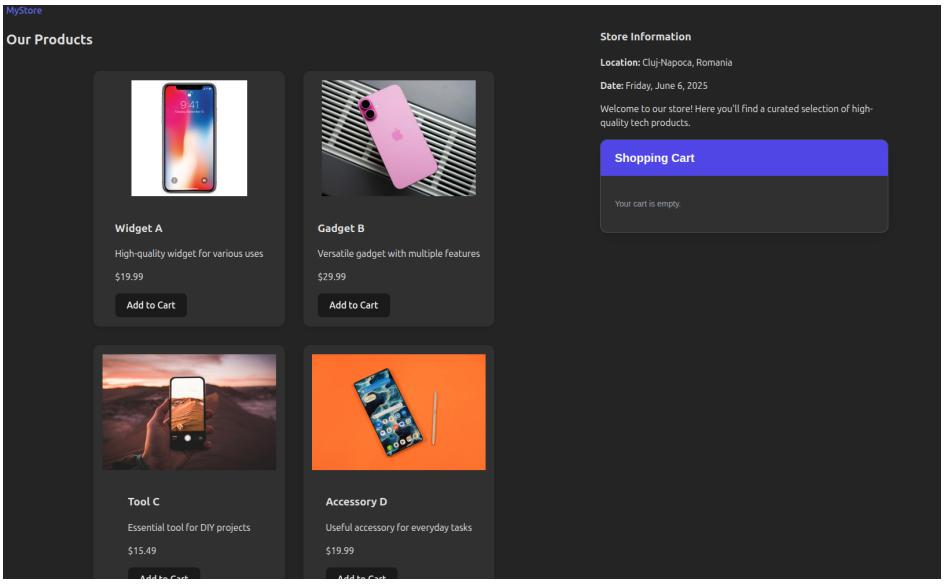

Figure 5.3: Product catalog view, showcasing products in a grid layout with an "Add to Cart" button and a persistent Shopping Cart module on the right.

each card, inviting interaction. On the right-hand side of the page, a persistent "Shopping Cart" module provides an at-a-glance summary of the items the user has selected. On the initial visit, this cart correctly indicates that it is empty. Figure 5.3

- Adding Items to the Cart: When a user clicks the "Add to Cart" button for an item, the interface provides immediate visual feedback. The selected product is instantly added to the Shopping Cart module on the right. The cart dynamically updates to show each unique item, its quantity (which can be incremented), and its price. A subtotal, calculated tax, and a final, bolded total are displayed in real-time. This seamless and interactive experience allows users to track their selections and total cost without navigating to a separate page. Figure 5.4

- Checkout Process: Once the user has finished shopping, they click the "Proceed to Checkout" button located at the bottom of the shopping cart. This action signifies the handoff from browsing to payment and redirects the user away from the application's storefront to a secure, externally hosted payment page powered by Stripe. Figure 5.5

- Stripe Payment Page: The application's integration with Stripe Checkout is seamless. On the secure Stripe page, the user sees a final order summary on the left, itemizing the products and the total charge. On the right, they are presented with a standard, secure form to enter their email and credit card information. This critical design choice offloads the responsibility of handling

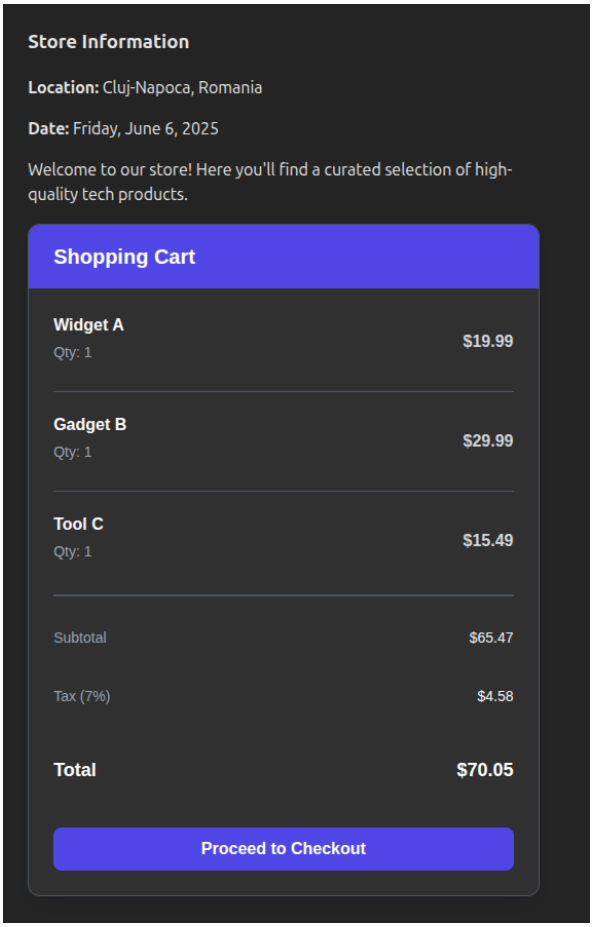

Figure 5.4: Shopping Cart view, showing selected products with quantities, prices, and a total cost. The ”Proceed to Checkout” button is highlighted at the bottom.

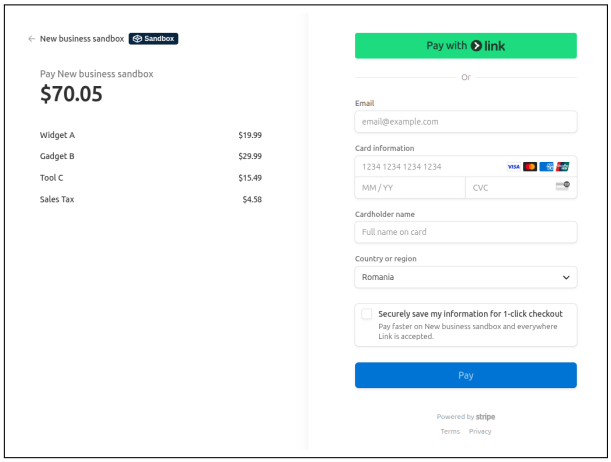

Figure 5.5: Stripe Payment Page, displaying a final order summary on the left and a secure form for entering payment details on the right. [Str25]

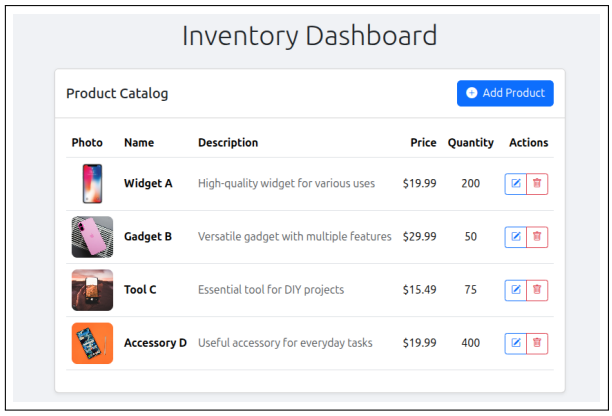

Figure 5.6: Inventory Dashboard, displaying a table of products with options to edit, delete, or add new products. Each product row includes a photo, name, description, price, and quantity.

and securing sensitive payment card industry (PCI) data to a trusted, compliant third-party provider, which is a best practice that significantly enhances the security and trustworthiness of the application. [Str25] Figure 5.5

5.5.2 Administrative Interface

A separate, secure web application serves as the administrative backend, allowing store managers to perform essential inventory management tasks.

- Inventory Dashboard: The primary administrative screen is the ”Inventory Dashboard.” It features a clean, tabular view of the entire ”Product Catalog.” Each row in the table corresponds to a single product, clearly displaying its photo, name, description, price, and the current quantity in stock. To facilitate easy management, each row includes ”Actions” buttons—an icon for editing and an icon for deleting the product. At the top of the catalog, an ”Add Product” button allows for the creation of new inventory items. Figure 5.6

- Editing a Product: When an administrator clicks the edit icon for a product, a modal form titled ”Edit: [Product Name]” appears, overlaying the dashboard. This form is pre-populated with all the existing product data (name, description, photo URL, price, and quantity). The administrator can modify any of these fields and then click the ”Update Product” button to save the changes to the database or ”Cancel” to discard them. This modal approach provides a quick and efficient workflow for making targeted updates to product details and stock levels without losing the context of the main dashboard. Figure 5.7

- Adding a New Product: Clicking the ”Add Product” button opens a similar modal form, but this one is empty, allowing the administrator to enter all nec-

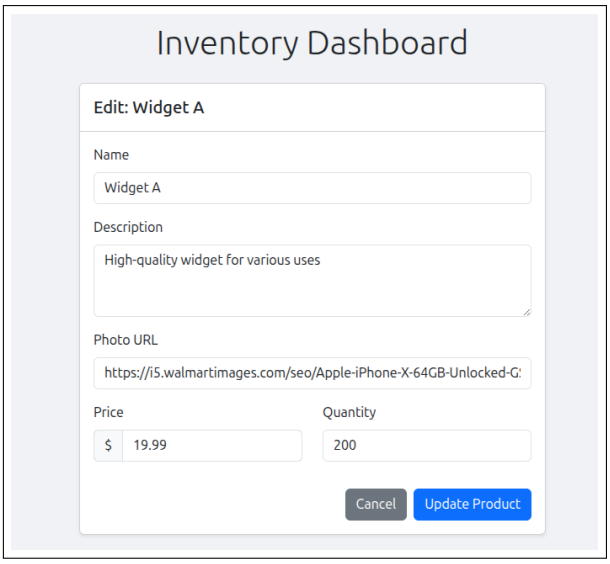

Figure 5.7: Edit Product Modal, allowing administrators to modify product details. The form is pre-populated with existing data, and changes can be saved or canceled.

essary details for a new product. After filling out the form, they can click "Create Product" to add it to the catalog or "Cancel" to close the modal without saving. Figure 5.7

- Deleting a Product: The delete action is straightforward. When the administrator clicks the delete icon for a product, a confirmation dialog appears, asking if they are sure they want to delete the product. This step prevents accidental deletions and ensures that the administrator has a chance to confirm their intent.

This clear separation of concerns between a public-facing storefront and a secure administrative backend is a standard and effective design pattern. It ensures a simple, intuitive shopping experience for customers while providing powerful and secure tools for store managers to maintain the product catalog.

Chapter 6

Results and Discussion

To rigorously validate the sustained performance and stability of the architecture, a 30-minute load test was conducted against the InventoryService. The test, orchestrated by Locust [Loc25], simulated a continuous load of 10 concurrent users (6.3) making GET requests to the /inventory endpoint. The health and resource consumption of the service were closely monitored throughout the test using Grafana dashboards populated with data from Prometheus.

6.0.1 Results

The 30-minute sustained load test yielded the following key performance indicators from Locust:

- Requests per Second (RPS): After an initial ramp-up, the system achieved and maintained a remarkably stable throughput, averaging approximately 124 RPS. This rate was held consistently for the entire 30-minute duration, demonstrating the service's ability to handle a continuous load without degradation. Figure 6.1
- Response Times: The service exhibited excellent and predictable latency. The median (50th percentile) response time was stable at approximately 35ms, while the 95th percentile response time held steady at around 50ms. The absence of any upward trend in response times over the 30-minute period confirms the efficiency of the service. Figure 6.2

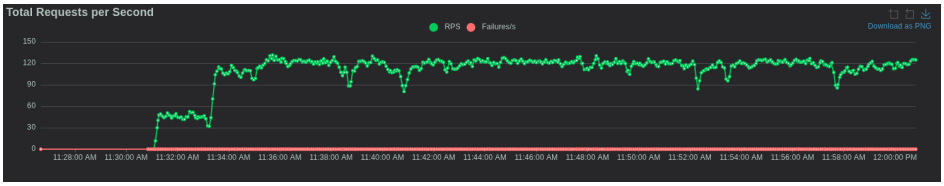

Figure 6.1: Requests per Second (RPS) during the 30-minute load test.

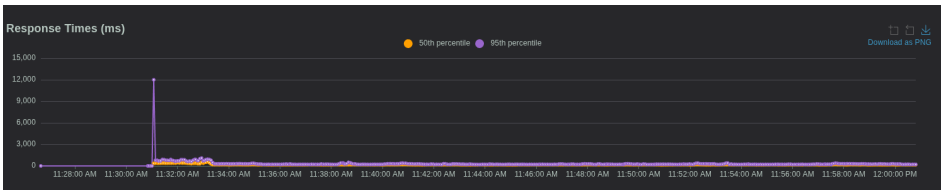

Figure 6.2: Response times during the 30-minute load test, showing median and 95th percentile latencies.

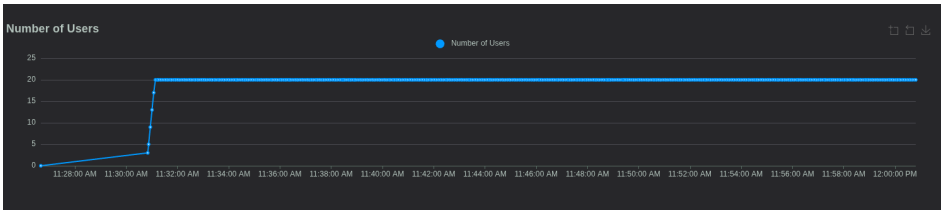

Figure 6.3: Number of users during the 30-minute load test, showing a stable count of 10 concurrent users.

- Failures: The system demonstrated high reliability, with zero failures recorded across the entire 30-minute test. This 100% success rate under sustained load confirms the robustness of the service.

The Grafana dashboard, capturing a 30-minute window, provided crucial insights into the behavior of the InventoryService pod within Kubernetes:

- CPU Usage: At the start of the test, the CPU usage of the primary inventory-service pod rapidly climbed from its idle state to its configured limit of 0.20 cores (200m). The pod sustained this level of maximum CPU utilization for the entire test period.Figure 6.4
- Memory Usage: Memory consumption remained exceptionally stable at approximately 97 MB. The graph shows a flat line, indicating no memory leaks or gradual resource exhaustion over the extended test duration. Figure 6.5

6.0.2 Discussion

The results from this 30-minute load test provide strong evidence of the InventoryService's stability, performance, and the effectiveness of its cloud-native design.

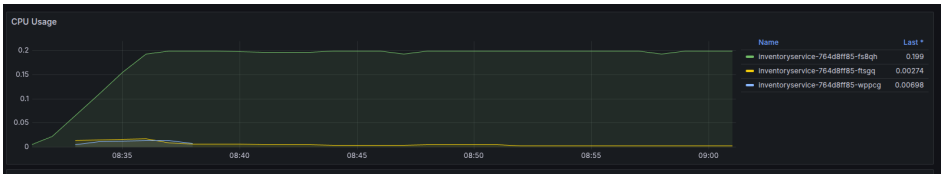

Figure 6.4: CPU usage of the InventoryService pod during the 30-minute load test, showing sustained maximum utilization.

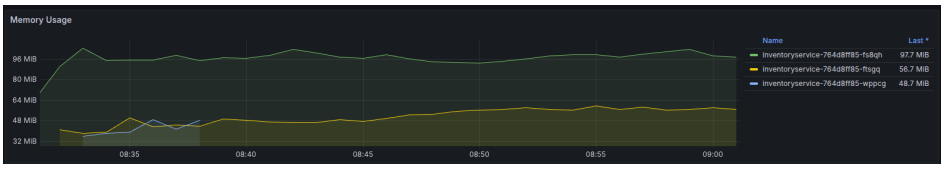

Figure 6.5: Memory usage of the InventoryService pod during the 30-minute load test, showing stable memory consumption.

Sustaining 124 RPS with a median latency of 35ms for 30 minutes is a significant achievement. It proves that the service is not only fast but also efficient, without performance degradation over time. The flat memory usage graph is particularly noteworthy, as it confirms the absence of memory leaks, a common issue that plagues long-running services and often leads to crashes.

The most critical insight is the interplay between the application's performance and the Kubernetes orchestration layer. The CPU utilization for the single inventoryservice pod was consistently pegged at its limit. This behavior under sustained load validates two core architectural tenets:

1. The Service is CPU-Bound: The performance of the service under this workload is primarily constrained by its available processing power. This is a predictable and desirable behavior, as it means performance can be directly influenced by resource allocation.

2. Validation of the Horizontal Scaling Trigger: The Horizontal Pod Autoscaler (HPA) is configured to provision a new pod when the average CPU utilization exceeds 80%. Since the pod's CPU was held at nearly 100% of its limit, it continuously met the condition for scaling out. This test proves that under a real-world scenario with increasing load, the HPA would be correctly triggered to horizontally scale the service. This ensures the system can elastically adapt to demand, maintaining low response times by distributing the load across multiple instances.

In summary, the 30-minute endurance test demonstrates more than just performance; it confirms the system's operational stability and resilience. The InventoryService has proven to be a robust, efficient, and well-behaved component of the architecture, whose behavior under load validates the core principles of the automated horizontal scaling strategy designed to ensure high availability and a seamless user experience.

Chapter 7

Conclusion and Future Research Directions

7.1 Summary of Key Findings and Contributions

This thesis has undertaken a comprehensive exploration of scalability and fault tolerance within the domain of microservice architectures. The analysis reveals that achieving these critical system qualities is not an incidental outcome of adopting microservices but requires a deliberate, multi-faceted architectural approach. Key findings underscore that robust microservice systems are built upon a foundation of well-chosen design patterns, enabling technologies, and proactive resilience engineering practices.

The shift from monolithic to microservice architectures offers inherent advantages for targeted scalability and improved fault isolation. However, the distributed nature of microservices introduces new complexities, including challenges in inter-service communication, data management, and operational overhead. This research has systematically examined strategies to address these challenges. For scalability, patterns such as horizontal scaling, load balancing, API gateways and asynchronous processing via message queues have been identified as crucial. For fault tolerance, patterns including circuit breakers advantages intelligent retries with backoff are essential for building resilient systems.

The role of enabling technologies cannot be overstated. Containerization with Docker and orchestration with Kubernetes (including its diverse autoscaling mechanisms like HPA) provide the platform for dynamic resource management. Furthermore, comprehensive observability—through monitoring and logging emerges as the bedrock upon which effective scaling and fault diagnosis are built. Proactive resilience practices represent a paradigm shift towards anticipating and mitigating failures before they impact users.

The primary contribution of this thesis is the design, end-to-end implementation, and empirical validation of a holistic architectural framework for microservices. This framework was brought to life through a concrete e-commerce application built with .NET and deployed on Kubernetes, demonstrating how to effectively integrate a suite of modern cloud-native technologies.

Specifically, this work established a blueprint for:

- Independent Scalability: Leveraging Kubernetes and the Horizontal Pod Autoscaler, we proved how services can scale automatically based on real-time CPU demand.
- Layered Fault Tolerance: Resilience was woven into the architecture at multiple levels—from application-level Retry and Circuit Breaker patterns using Polly to infrastructure-level self-healing via Kubernetes and guaranteed message delivery through RabbitMQ.
- Decoupled Communication: Asynchronous messaging was employed to decouple the Order, Inventory, and Payment services, enhancing both resilience and scalability.
- Comprehensive Observability: A full monitoring stack using Prometheus and Grafana was integrated, providing crucial, real-time visibility into system performance and resource utilization under load.

Crucially, this thesis moved beyond theoretical design by subjecting the system to rigorous load testing with Locust. The results provided empirical evidence that the InventoryService is stable, performant, and CPU-bound under sustained load, validating that the configured auto-scaling policies would be triggered under real-world conditions.

Ultimately, this thesis argues and demonstrates that scalability and fault tolerance are deeply symbiotic. The successful integration of the specific patterns and technologies presented here—where observability informs scaling, and resilience patterns protect services during that scaling—is key to developing microservice applications that are not only adaptable to dynamic demands but are verifiably durable in the face of inevitable failures.

7.2 Limitations of the Study

It is imperative to recognize certain limitations inherent in this study. The rapid pace of innovation in microservices and their associated technologies means that some of the specific tools and performance metrics presented herein may have a limited shelf

life. Furthermore, the empirical validation was conducted on a specific e-commerce platform within a controlled setting; consequently, the generalizability of these findings to other domains or application types is not guaranteed. Finally, the scope of this thesis was intentionally focused on scalability and fault tolerance, thereby excluding an in-depth analysis of other vital microservice considerations such as security, data consistency, and the developer experience, which remain fruitful avenues for future research.

7.3 Future Research Directions

The dynamic and complex nature of microservice architectures presents numerous opportunities for future research. Several promising avenues include:

- AI and Machine Learning for Intelligent System Management [Wal25] [Mak23]
- Advanced Chaos Engineering and Resilience Verification [Gre23]
- Serverless Architectures and Function-as-a-Service (FaaS)
- Dynamic Microservice Architectures [BIG19] [IA22]

Addressing these research areas will contribute to the continued evolution of microservice architectures, enabling the development of even more scalable, resilient, and intelligent distributed systems.

Bibliography

[BGO+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[BIG19] Mihai Baboi, Adrian Iftene, and Daniela Gîfu. Dynamic microservices to create scalable and fault tolerance architecture. *Procedia Computer Science*, 159:1035–1044, 2019.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[FL14] Martin Fowler and James Lewis. Microservices: a definition of this new architectural term, 2014. Online; accessed 15 June 2025.

[Gra25] Grafana Labs. Grafana: The open observability platform, 2025. Online; accessed 15 June 2025.

[Gre23] Gremlin. Chaos engineering: The history, principles, and practice, 2023. Online; accessed 15 June 2025.

[IA22] Abdullahi Ibrahim and Mallo Ade. Scalable fault tolerance for microservices-based systems. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 6329–6331, 2022.

[Kub25] Kubernetes Documentation. Horizontal pod autoscaler, 2025. Online; accessed 15 June 2025.

[Loc25] Locust Maintainers. Locust: Open source load testing tool, 2025. Online; accessed 15 June 2025.

[Mak23] Chisenga Makungu. Fault tolerance in distributed systems. *World Journal of Innovation and Modern Technology*, 7(2):105–108, dec 2023.

[Nyg18] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

[Pol25] Polly. Polly: The .net resilience library, 2025. Online; accessed 15 June 2025.

[Pro25] Prometheus Authors. Prometheus - monitoring system & time series database, 2025. Online; accessed 15 June 2025.

[Rab25] RabbitMQ. Messaging with rabbitmq, 2025. Online; accessed 15 June 2025.

[Ric18] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning Publications, 2018.

[She24] Gaurav Shekhar. Microservices design patterns for cloud architecture. *International Journal of Computer Science and Engineering*, 11(9):1–7, sep 2024.

[Str25] Stripe. Stripe documentation: Online payments, 2025. Online; accessed 15 June 2025.

[Wal25] Anjali Walia. Leveraging microservices architecture: Key principles for scalable and fault-tolerant applications. *International Journal of Information Technology and Management Information Systems*, 16(1):455–468, feb 2025.