Yashwant_Kundathil-MSc_Research_Proposal.pdf

by Yash Kundathil

Submission date: 25-Feb-2025 06:27PM (UTC+0000)
Submission ID: 251211769
File name: Yashwant_Kundathil-MSc_Research_Proposal.pdf (135.66K)
Word count: 3289
Character count: 20533

Design and Evaluation of a Fault-Tolerant Library for Automated Request Auditing and Server-Side Resilience in Distributed Systems.

Yashwant Kundathil

Research Proposal

February 2025

Abstract

This research proposes the design and evaluation of a fault-tolerant library aimed at enhancing server-side resilience in distributed systems. The library introduces an automated mechanism to audit all HTTP requests, categorizing them as successful or failed based on server response codes. For server-side exceptions, the library enables developers to process failed requests through configurable options, such as periodic custom logic or batch processing. Additionally, it automates the creation of required database tables, simplifying integration and reducing developer overhead. The study evaluates the library's performance under two configurations: using an external database (Postgres) and an in-memory database (H2). Key metrics such as latency, throughput, and resource utilization are assessed. The proposed solution is benchmarked against existing methodologies, including Circuit Breakers, Bulkheads, and Retries with exponential backoff. By addressing critical gaps in fault tolerance mechanisms, this research contributes to the reliability, maintainability, and scalability of microservices architectures.

Keywords: Fault Tolerance, Distributed Systems, Server-Side Resilience, H2, Postgres

Table of Contents

Background

Modern microservices often rely on stateless interaction patterns and distributed components, but they still face reliability issues due to partial failures and inconsistent request handling (Song and Li, 2024). Particularly, microservices architectures with numerous HTTP-based endpoints may encounter transient or persistent errors that lead to lost or partially processed data (Montesi and Weber, 2016). As more services coordinate via REST or similar protocols, even a minor request failure can have cascading effects, as seen in microservice fault tolerance studies by (Larsson et al., 2021).

To address these concerns, researchers have proposed a variety of fault tolerance and retry strategies (Vale et al., 2022). These approaches can reduce data loss, but in many cases do not provide a structured auditing mechanism to capture all HTTP request outcomes for subsequent analysis (Gu et al., 2011). While general microservices guidance (Thönes, 2015) encourages the use of logs or queue-based backbones, reactive solutions usually revolve around circuit breakers, retries, or fallback routines (Rasheedh and Saradha, 2021). However, such approaches may not maintain a fine-grained log of request success or failure codes unless explicitly configured.

Aspect-Oriented Programming (AOP) has emerged as a suitable paradigm for inserting cross-cutting logic (such as monitoring and auditing) into existing services without scattering fault-tolerance code across every module (Kiczales et al., 1997) (Walker, Zdancewic, and Ligatti, 2003). By weaving auditing and retry aspects at key join points, one can systematically track request completions and failures (Amazon Web Services (AWS), 2019) (Paul, Jagnani, and Supraja, 2023). This pattern is especially helpful in building self-healing systems: for instance, a partial reprocessing engine can re-run failed HTTP transactions after a waiting period, akin to how queue systems reattempt unacknowledged messages (Sreekanti et al., 2020).

Further, advanced serverless or workflow-based solutions show that it is possible to ensure robust transactional execution, including auditing, through specialised libraries that handle distributed state (Zhang et al., 2020). In these systems, the logging or journaling layer captures all function or request outcomes, thereby enabling replays or targeted reprocessing. Although these serverless fault-tolerant frameworks (Barrak, Petrillo, and Jaafar, 2023) emphasise large-scale reliability, they do not specifically cover how to systematically audit all HTTP-based failures in a microservice context and re-run them as needed.

Hence, there is a gap in designing a more direct library-driven approach where every HTTP request is captured, classified (by its status code), and then automatically retried at a later interval if it fails. Such an auditing library, built with AOP techniques, can attach to microservices seamlessly, injecting logic for logging each request's outcome (including success or failure code), generating "audit tables," and scheduling reprocessing steps. This is similar to the "dead letter queue" patterns recommended for event-driven architectures (Amazon Web Services (AWS), 2019) (Paul, Jagnani, and Supraja, 2023) but would focus on HTTP requests specifically and integrate with established microservice libraries.

By bridging AOP-based cross-cutting logging with robust auditing–reprocessing patterns, we can create a self-healing microservices environment: the system automatically records every request outcome, determines failures, and triggers reprocessing after a configurable wait. Such a design could mitigate the "silent failures" that often plague distributed HTTP traffic, thus improving end-to-end reliability and transparency in microservices.

Related Work

Research on improving fault tolerance and reliability in microservices has expanded, driven by the rising complexity of distributed applications and the need to address HTTP-level failures. This literature review examines work that focuses on retry policies, aspect-oriented programming (AOP), dead-letter queues, transactional workflows, and advanced serverless or peer-to-peer (P2P) techniques.

Fault Tolerance and Retry Strategies in Microservices

(Song and Li, 2024) closely examines different retry approaches specifically for microservice applications. Their analysis reveals how excessive retries can result in "retry storms" and inadvertently amplify latencies if not tuned correctly. This study supports the concept of carefully auditing failed HTTP requests rather than reissuing them blindly, aligning well with the proposed auditing idea.

Additionally, (Montesi and Weber, 2016) discusses microservices patterns such as circuit breakers and API gateways, emphasising the importance of short-circuiting upstream failures. While circuit breakers are effective for resilience, they do not inherently provide a structured way to re-run failed HTTP calls. In related work, (Larsson et al., 2021) propose "soft circuit breakers" in service meshes, using cached responses to mitigate total outages. However, they focus on dynamic load adaptation rather than capturing and replaying each HTTP request failure.

Aspect-Oriented Programming (AOP) for Reliability

(Kiczales et al., 1997; Walker, Zdancewic, and Ligatti, 2003) describe how AOP can separate cross-cutting concerns like logging, monitoring, or retry logic from core service code. By weaving auditing logic at relevant join points, engineers can systematically track whether each HTTP request succeeded or failed. This modular approach avoids scattering error-handling logic in every microservice endpoint, which aligns with the creation of an auditing and reprocessing library.

Dead-Letter Queues and Their Role in Fault Handling

Dead-letter queue concepts have been explored extensively in serverless contexts. For instance, (Amazon Web Services (AWS), 2019; Paul, Jagnani, and Supraja, 2023) illustrate how queueing failed messages helps avoid data loss and allows developers to investigate and replay "poison messages." Although these references focus largely on asynchronous event-driven systems, the main principle of capturing failures and eventually reprocessing them is highly relevant to microservices that use HTTP.

Transactional and Serverless Approaches

To ensure strong consistency, (Sreekanti et al., 2020) introduce a shim that interposes between a Functions-as-a-Service platform and the underlying storage layer, enforcing read-atomic isolation. Although the emphasis is on transactional correctness, the notion of blocking partial writes until verified echoes the idea of re-running or validating partially executed requests.

(Zhang et al., 2020) present "Beldi," enabling transactional behaviour for serverless functions via log-based replays. Their system can re-run incomplete tasks and preserve exactly-once semantics. However, they do not specifically focus on logging each HTTP request code and reprocessing just the failures. (Barrak, Petrillo, and Jaafar, 2023), in contrast, propose a peer-to-peer approach for distributed machine learning using serverless architecture. They show that carefully designed logging and checkpointing can reduce partial updates and failures. While aimed at ML workflows, the approach of systematically recording state to improve reliability is consistent with auditing and replay for microservice HTTP calls.

Patterns and Empirical Studies on Microservices

(Thönes, 2015) offers a concise overview of microservices, underlining their modularity but also highlighting reliability issues in multi-service workflows. (Shekhar, 2024) details microservice design patterns such as bulkheads, circuit breakers, and retries, while (Vale et al., 2022) empirically examines how these patterns can affect maintainability versus performance. Their research suggests that crosscutting reliability logic sometimes complicates code; thus, an AOP-based solution can reduce code scattering and preserve maintainability.

Workflow Systems and Retrying

Finally, (Gu et al., 2011) investigates distributed workflows handling streaming data. They emphasise that naive fault handling may cause repeated or partial deliveries of data. Their recommendation for more thorough logging or checkpointing resonates with the proposed idea of building a library that audits every HTTP request attempt and systematically replays failed ones.

Summary of Gaps Identified

Although various retry, circuit breaker, and queue-based methods exist for microservices fault tolerance, they typically lack a unified system for auditing each HTTP request result code, labelling requests as successful or failed, and scheduling a re-run after a configured interval. Aspect-oriented techniques (Kiczales et al., 1997; Walker, Zdancewic, and Ligatti, 2003) demonstrate how cross-cutting logic might be weaved in, yet existing literature has not emphasised a simple library-based approach with "audit tables" and reprocessing integrated via AOP. The research proposed here attempts to fill that gap by drawing on concepts of systematic logging from queue-based and transactional designs, merging these with AOP modules and microservices best practices.

Research Questions

Research Questions

RQ1. *How effectively can the proposed library categorize HTTP requests based on server and non-server response codes to improve fault detection?*

RQ2. *What are the performance trade-offs between using an external database (Postgres) and an in-memory database (H2) for auditing requests?*

RQ3. *How does the proposed fault-tolerance mechanism compare in performance and reliability to existing methodologies such as Circuit Breakers, Bulkheads, and Retries with exponential backoff?*

Aim and Objectives

Aim

To develop a comprehensive fault tolerance library for distributed systems, enabling automated auditing and processing of failed HTTP requests, with configurable options for developers to manage failures either through periodic custom logic or batch processing, thereby enhancing system reliability and resilience.

Objectives

1. To design and implement an auditing mechanism to track all HTTP requests and responses, automatically recording them in an audit table.
2. To develop a database table creation module that seamlessly integrates with the library to automate the creation of required tables during deployment.
3. To introduce configurable failure processing options, allowing developers to:
   - Implement custom logic for periodic retries of failed requests.
   - Execute batch processing for all failed requests in a single run.
4. To evaluate the library's performance and scalability, focusing on metrics such as fault recovery time, system throughput, and impact on database operations.

Significance of the Study

1. The proposed library automates the classification of HTTP requests and provides flexible options for reprocessing failed requests. This reduces the need for manual fault-handling and lowers the chances of human errors, improving the speed of recovery when a service fails.
2. Adding fault-tolerance mechanisms directly into server-side code helps ensure system availability, even when there are temporary or long-term failures in downstream services. The library improves this process by allowing failed requests to be batched and reprocessed, overcoming the limitations of traditional retry mechanisms.
3. The automatic creation of database tables, along with configurable fallback options, reduces the need for developers to manually implement monitoring, logging, and recovery mechanisms. This lowers the cognitive load and ensures that fault-handling practices remain uniform across different microservices.
4. Organising failure detection and reprocessing within a single library makes the codebase easier to manage. As systems grow in complexity, having a centralised approach to fault handling allows services to scale efficiently with minimal modifications.

5. The proposed library works alongside existing fault tolerance techniques such as circuit breakers, retries, and bulkheads. This makes it easier for development teams to incorporate standard resilience patterns into their microservices in a structured and automated manner.

Scope of the Study

1. Library Development :: This study focuses on designing and developing a reusable library that automates the creation of database tables for auditing failed HTTP POST requests. The library includes built-in logic to record, classify, and reprocess these failed transactions.

2. Service Architecture :: Two services are central to this work.

   - Service A (Client Service): Sends HTTP POST requests and requires only basic configuration, such as database details and retry intervals.
   - Service B (Database-Connected Service): Connects to an in-memory or external database to store and manage the audit tables created by the library.

3. Configurable Reprocessing :: Once a failed request is recorded in the audit table, it is reprocessed automatically after a predefined time. Users can configure settings such as the retry interval and batch size to meet different application needs.

4. Database Variants :: The study evaluates the performance implications of two distinct database setups for storing failed requests:

   - In-Memory Database (H2): Prioritizes faster writes and reads but may have constraints on capacity and persistence.
   - External Database (Postgres): Offers robust persistence and scalability at the potential expense of higher latency.

5. Comparison with Existing Fault Tolerance Mechanisms The proposed library will be benchmarked against established fault-tolerance solutions, including:

   - Circuit Breakers (to isolate failing services),
   - Retry Policies (with exponential backoff),
   - Bulkheads (to segment resources and minimize the blast radius of failures).

   This study aims to evaluate the effectiveness of the proposed auditing and reprocessing library in a real-world microservices environment. By testing it under different storage configurations and comparing it with established fault-tolerance mechanisms, the research will assess its ability to enhance system reliability and maintainability.

Research Methodology

Experiment 1 : Evaluating In-Memory VS External Database Configuration

Experiment Objective The purpose of this experiment is to compare the performance and resource utilization of two different configurations for storing audit data in a microservices architecture:

1. **In-Memory Database (H2):** ServiceB uses an in-memory database to store audit logs.
2. **External Database (PostgreSQL):** ServiceB uses an external database to store audit logs.

We aim to determine which configuration offers better performance, efficiency, and reliability under different operational conditions.

Experimental Setup

We have a system consisting of two microservices:

1. **ServiceA:**
    - Connects to a database that stores user-related information.
    - Responsible for receiving user data from ServiceB and persisting it to the database.
2. **ServiceB:**
    - Receives user data from external sources.
    - Sends this data to ServiceA for storage.
    - Utilizes the audit framework to log failed requests when ServiceA is unavailable.

We will conduct two main tests under each configuration (H2 and PostgreSQL) ::

1. Load Test with Both Services Running :: Ensure both ServiceA and ServiceB are operational and ensure all the requests are getting audited and marked as successful. Perform load test on ServiceB with 10K+ requests simulating incoming data.
2. Load Test when ServiceA is down :: Shut down ServiceA to simulate a service outage. Perform load test on serviceB with 10K+ requests. All the requests should be audited and marked as failed transactions.

Metrics to Evaluate:

- **Completion Time:** Total time taken to process all requests.
- **Memory Utilization:** Amount of memory used by ServiceB during the test.
- **Response Time:** Average time ServiceB takes to respond to each request.
- **Data Persistence:** Verify that all user data is successfully stored in ServiceA’s database.

Experiment 2 : Evaluating novel framework against traditional frameworks

Evaluating the novel framework and how it helps to isolate errors and reprocessing without losing requests and how it compares to traditional approaches like ::

1. Bulk Head
2. Retries Exponential Backoff
3. Circuit Breaker

Experimental Setup

We have a system consisting of two microservices:

1. **ServiceA:**
   - Connects to a database that stores user-related information.
   - Responsible for receiving user data from ServiceB and persisting it to the database.
2. **ServiceB:**
   - Receives user data from external sources.
   - Sends this data to ServiceA for storage.
   - ServiceB will be used to validate how the fault tolerance framework works with different retry mechanisms like Circuit Breaker, Retries with exponential backoffs and BUlkhead.

We will bring down serviceA and compare how traditional frameworks work when compared to Novel framework as suggested in this study.

Metrics to Evaluate:

- **Memory Utilization:** Amount of memory used by ServiceB during the test.
- **Response Time:** Average time ServiceB takes to respond to each request.
- **Data Persistence:** Verify that all user data is successfully stored in ServiceA's database.

Requirements Resources

The implementation and evaluation of the proposed fault tolerance library require the following resources:

Software and Tools

- **Java Development Kit (JDK)**: Version 17 for library development and testing.
- **Spring Boot Framework**: Version 3.0.1 for creating and integrating microservices.
- **Databases**:
  - PostgreSQL: Version 15.2 for external database evaluations.
  - H2 Database: Version 1.4.200 for in-memory database evaluations.
- **Testing Tools**:
  - Postman: For API testing and simulating failures.

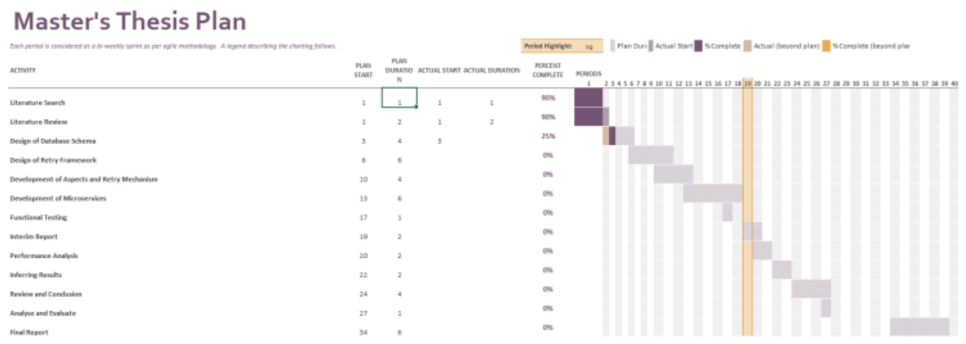
Hardware Requirements

- Personal laptop or desktop with the following specifications:
  - Processor: Intel Core i3 or equivalent.
  - RAM: 8 GB or higher.
  - Storage: Minimum 500 GB SSD.
- Cloud infrastructure (optional): For simulating large-scale microservices environments.

Development Environment

- Integrated Development Environment (IDE): IntelliJ IDEA or Eclipse with Maven for project management.
- Version Control: Git for tracking and managing code changes.

Research Plan



Master's Thesis Plan

Each period is considered as a bi-weekly sprint as per agile methodology. A legend describing the charting follows.

Period Highlight: 19

▨ Plan Dur | ▨ Actual Start | ■ % Complete | ▨ Actual (beyond plan) | ■ % Complete (beyond plan)

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Literature Search	1	1	1	1	90%
Literature Review	1	2	1	2	90%
Design of Database Schema	3	4	3		25%
Design of Retry Framework	6	6			0%
Development of Aspects and Retry Mechanism	10	4			0%
Development of Microservices	13	6			0%
Functional Testing	17	1			0%
Interim Report	19	2			0%
Performance Analysis	20	2			0%
Inferring Results	22	2			0%
Review and Conclusion	24	4			0%
Analyse and Evaluate	27	1			0%
Final Report	34	6			0%

Figure 1: Gantt chart of the Master’s Thesis.

References

Amazon Web Services (AWS) (2019). *Durable Serverless Architectures: Working with Dead-Letter Queues*. Accessed: YYYY-MM-DD. URL: https://d1.awsstatic.com/events/reinvent/2019/Durable_serverless_architecture_Working_with_dead-letter_queues_API309.pdf.

Barrak, Amine, Fabio Petrillo, and Fehmi Jaafar (Feb. 2023). "Architecting Peer-to-Peer Serverless Distributed Machine Learning Training for Improved Fault Tolerance". In: *arXiv preprint*. arXiv: 2302.13995 [cs.DC]. URL: http://arxiv.org/abs/2302.13995.

Gu, Yi et al. (2011). "Improving throughput and reliability of distributed scientific workflows for streaming data processing". In: *Proc.- 2011 IEEE International Conference on HPCC 2011 - 2011 IEEE International Workshop on FTDCS 2011 -Workshops of the 2011 Int. Conf. on UIC 2011- Workshops of the 2011 Int. Conf. ATC 2011*, pp. 347–354. ISBN: 9780769545387. DOI: 10.1109/HPCC.2011.52.

Kiczales, Gregor et al. (1997). *Aspect-Oriented Programming Aspect-Oriented Progra m ming*.

Larsson, Lars et al. (2021). *Towards Soft Circuit Breaking in Service Meshes via Application-agnostic Caching*. arXiv: 2104.02463 [cs.NI]. URL: https://arxiv.org/abs/2104.02463.

Montesi, Fabrizio and Janine Weber (Sept. 2016). "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *arXiv preprint*. arXiv: 1609.05830 [cs.PL]. URL: http://arxiv.org/abs/1609.05830.

Paul, Gita Alekhya, Yashvardhan Jagnani, and P. Supraja (2023). "Implementing Dead Letter Exchanges in MQTT and Proposing a Broker Failure Algorithm Utilizing Blockchain DNS". In: *ViTECoN 2023 - 2nd IEEE International Conference on Vision Towards Emerging Trends in Communication and Networking Technologies, Proceedings*. Institute of Electrical and Electronics Engineers Inc. ISBN: 9798350347982. DOI: 10.1109/ViTECoN58111.2023.10157491.

Rasheedh, J Abdul and S. Saradha (Aug. 2021). "Reactive Microservices Architecture Using a Framework of Fault Tolerance Mechanisms". In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pp. 146–150. DOI: 10.1109/ICESC51422.2021.9532893.

Shekhar, Gaurav (Sept. 2024). "Microservices Design Patterns for Cloud Architecture". In: *Int. J. Comput. Sci. Eng.* 11.9, pp. 1–7.

Song, Alan and Lisa Li (May 2024). *Characterizing Retry Policies for Microservice Applications*.

Sreekanti, Vikram et al. (Apr. 2020). "A fault-Tolerance shim for serverless computing". In: *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*. Association for Computing Machinery, Inc. ISBN: 9781450368827. DOI: 10.1145/3342195.3387535.

Thönes, Johannes (2015). *Software Engineering*. Accessed: YYYY-MM-DD. URL: http://www.ieee.org/publications_.

Vale, Guilherme et al. (Jan. 2022). "Designing Microservice Systems Using Patterns: An Empirical Study on Quality Trade-Offs". In: *arXiv preprint*. arXiv: 2201.03598 [cs.SE]. URL: http://arxiv.org/abs/2201.03598.

Walker, David, Steve Zdancewic, and Jay Ligatti (2003). *A Theory of Aspects*.

Zhang, Haoran et al. (2020). *Proceedings of the 14th USENIX conference on Operating Systems Design and Implementation (OSDI '20) : November 4–6, 2020*. USENIX Association. ISBN: 9781939133199.

Yashwant_Kundathil-MSc_Research_Proposal.pdf

ORIGINALITY REPORT

2%	1%	0%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Liverpool John Moores University Student Paper	1%
2	Submitted to University of Bedfordshire Student Paper	1%

Exclude quotes	Off	Exclude matches	< 1%
Exclude bibliography	On		