

Chapter 3

RESEARCH METHODS

3.1 Research Framework

This chapter outlines the systematic approach that used to investigate the research questions. It details the research design, the system environment, the procedure for data collection and analysis, and the tools used to conduct the experiment. The methodology is designed to be systematic and repeatable, ensuring the validity of the findings.

This study employs a quantitative, experimental research design. Specifically, it follows a pre-test/post-test methodology to assess the impact of targeted performance optimizations. The research framework, illustrated in Figure 3.1, follows a cycle of testing, analysis, and optimization that is repeated until performance goals are achieved.

3.2 Research Phases

The research process are divided into three primary phases, which form an iterative cycle:

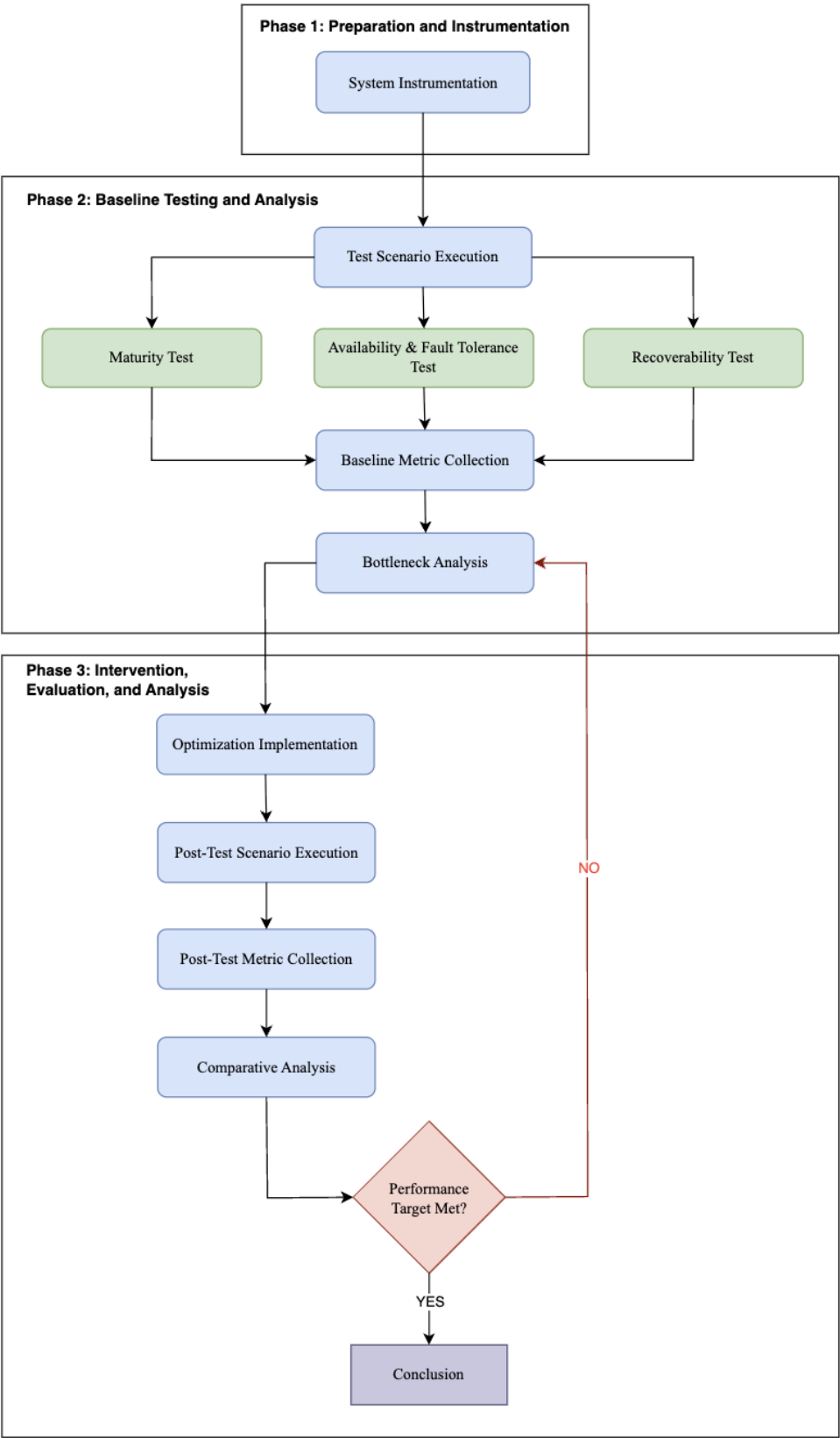


Figure 3.1: Research Framework

1. **Phase 1: Preparation and Instrumentation.** This foundational phase focuses

on setting up the necessary tools to achieve end-to-end system observability and meet the three pillars of observability (Logs, Metrics, Traces). The metrics are used for quantitative analysis of the performance bottleneck. Additionally, traces and logs provide the qualitative, diagnostic evidence to explain the findings and prove the optimizations work as intended.

2. **Phase 2: Baseline Testing & Analysis.** A series of load tests are conducted on the original system to establish a quantitative performance baseline. During this phase, performance bottlenecks are identified through the analysis of metrics and trace data.
3. **Phase 3: Intervention, Evaluation, and Analysis.** Based on the bottleneck analysis, a targeted optimization solution is implemented. Afterward, the exact same test scenarios are repeated on the optimized system to collect post-intervention data. The pre-test and post-test data are then comparatively analyzed. If performance goals are met, a conclusion is drawn. If not, the process returns to the Bottleneck Analysis step to address the next limiting factor, beginning a new iteration.

3.3 Test Scenarios and Data Generation

To provide a comprehensive assessment, the test scenarios are designed based on the reliability sub-characteristics from the ISO/IEC 25010 standard. To obtain the data for analysis, each test scenario was executed using k6, a high-performance load testing tool. The k6 scripts were written in JavaScript to simulate specific user behaviors and system stressors. While each scenario had a unique objective, a common baseline of system activity was simulated across all tests by having a portion of the Virtual Users (VUs) continuously send requests to a standard endpoint, such as *POST /api/contact*, to represent normal operational load. The specific execution method for each scenario is detailed below

1. **Maturity Test Scenario:** Maturity is how well the system operates under normal conditions and avoids failures due to bugs or design flaws. This scenario objective is to evaluate the system's stability when processing an unusually large and

resource-intensive request. A common flaw in data processing systems is the failure to handle edge cases related to data volume.

This scenario represents critical, batch-oriented business operations. For example, it simulates a year-end financial consolidation, the bulk import of a large product catalog from a supplier, or a one-time synchronization of thousands of employee records from an acquired company's HR system. In these situations, the data is valid, but its sheer volume can overwhelm an immature system, leading to resource exhaustion, memory leaks, and crashes

The execution of the scenario is a k6 script was configured to simulate two distinct concurrent activities, sustained normal load and "poison pill" injection. Sustained normal load is a group of 20 VUs was set to continuously send moderately sized requests to the *POST /api/contact* endpoint, while "poison pill" injection is executed during the normal load was running, the k6 script triggered a single, massive *POST* request to a product synchronization endpoint (*POST /api/product*). The body of this single request contained a JSON payload representing over 20,000 product records.

k6 collected the performance metrics (latency, throughput, error rate) for the sustained normal load. This data revealed if the "poison pill" request degraded the performance of unrelated, normal operations. Simultaneously, the observability stack (Prometheus and Jaeger) captured the monolith's resource consumption and traced the "poison pill" request's lifecycle, providing the data to confirm resource exhaustion or internal processing failures.

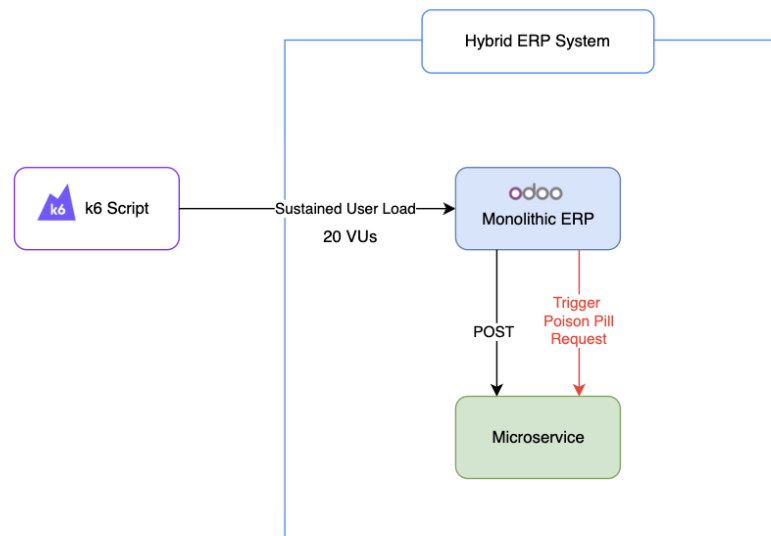


Figure 3.2: Diagram illustrates the Maturity Test for "poison pill" trigger.

2. **Availability & Fault Tolerance Scenario:** Availability is about the system being operational and accessible when required and Fault Tolerance is the system's ability to continue operating, even in degraded state, or after a component has failed completely. The scenario to evaluate the system's ability to isolate faults and prevent the failure of a single component from causing a cascading failure that impacts the entire system. Its purpose is to see if a fault in one component is properly contained or if it triggers a cascading failure, one of the most severe risks in such systems. A system that lacks fault isolation cannot be considered highly available.

The ISO/IEC 25010 split this 2 sub-category into separate characteristic, although in reliability engineering, these two concepts are distinct but deeply connected. Our scenario handle both characteristic by introducing slow microservice (a fault) to measure its availability in the presence of that fault, therefore, fault tolerance is the characteristic that enables high availability. A system that is not fault-tolerant cannot be highly available.

This scenario designed to simulates an extremely common operational problem. A downstream microservice might become slow due to a sudden traffic spike, a poorly optimized database query running long, or network latency to a third-party API (e.g., a payment gateway or shipping provider). A fault-tolerant system

must remain responsive for all other, healthy requests. This test determines if the monolith is "patient" to a fault (bad) or if it can "fail-fast" and protect itself (good).

This test required a preparatory step where a "sick" version of a downstream microservice was deployed, with its code intentionally modified to introduce a 15-second response delay. The k6 script was then executed with two parallel scenarios:

- (a) Fault inducing load: 20 VUs continuously sent requests to a monolith endpoint that depended on the "sick," delayed microservice.
- (b) Healthy Load: A separate group of 2 VUs sent requests to a simple, unrelated endpoint on the monolith (e.g., GET /api/contacts) that had no dependency on the slow microservice.

k6 collected and reported metrics for both scenarios independently. The critical data obtained was the response time for the "Healthy Load." A significant increase in its latency would quantitatively prove that the monolith's worker threads were being exhausted by the "sick" service, thus confirming a cascading failure.

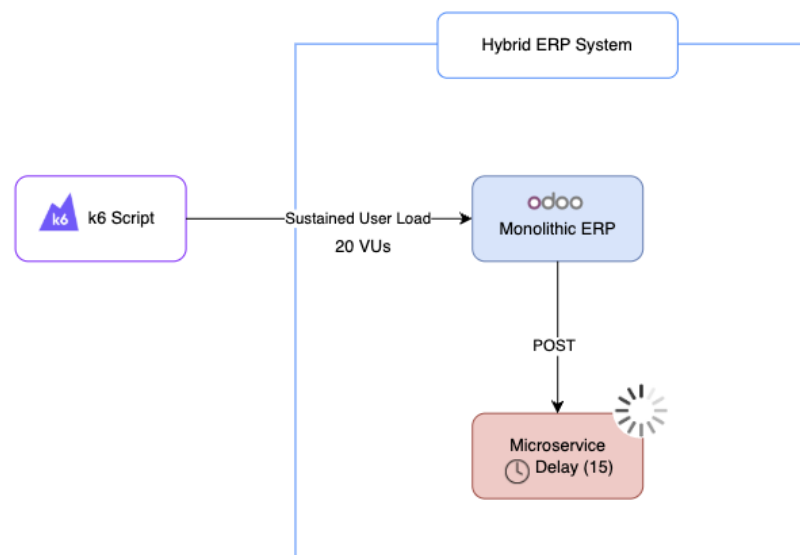


Figure 3.3: Diagram illustrate Availability & Fault Tolerance Test

3. **Recoverability Scenario:** Recoverability is the system's ability to re-establish a stable state and recover its data after a failure. The scenario designed to evaluate

the system's ability to re-establish a stable state after a complete component failure. To test the recoverability, k6 script will send sustained load to the monolith for 2 minutes, and then the target microservice is purposely shut down and then restarted after two minutes. The primary metric is the Time to Recover (TTR) and Failure Rate.

This test evaluates Recoverability by simulating a complete but temporary component failure. It distinguishes between infrastructure recovery (how fast a service can restart) and application recovery (whether the user even notices the outage). In modern cloud-native environments where services can restart automatically, the application's ability to handle this transient failure is a critical measure of its resilience.

This scenario directly mimics several common events: a microservice crashing due to a bug and being automatically restarted by its container orchestrator (like Docker); a planned "rolling update" deployment where a service is briefly unavailable; or a temporary network partition that makes a service unreachable for a short period. The goal is to see if the system masks this temporary outage from the user, for instance, by using a retry mechanism, thereby minimizing the impact of the failure.

This test required combination of a k6 script with a manual, timed intervention: sustained load and manual intervention. Sustained load initiated from k6 script run a continuous load of 20 VUs against the *POST /api/contact* endpoint for a multi-minute duration. While manual intervention is executed at the two-minute mark of the test, the Docker container of the target microservice was manually stopped using a *docker stop* command. After a two-minute outage, the container was restarted with *docker start*.

The k6 test continued to run throughout the intervention. Its logs provided a continuous timeline of request outcomes, showing exactly when requests started to fail (at the moment of shutdown) and when they started to succeed again (once the service was back online). This raw data was used to calculate the two primary metrics for this test: the total user-facing failure rate during the incident window and the Time To Recover (TTR).

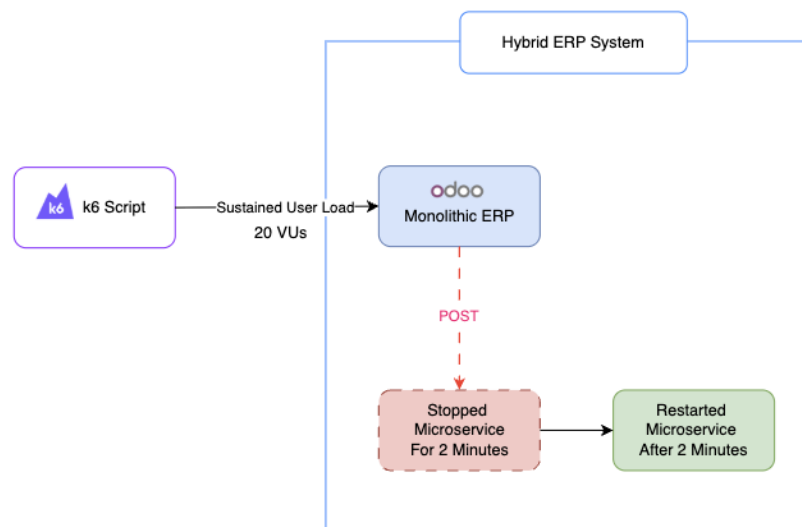


Figure 3.4: Diagram illustrate the Recoverability Test by shutting down and reboot the Microservice

3.4 Data Collection and Analysis

To ensure an objective evaluation, the following quantitative metrics are collected from the k6 test results for both the pre-test and post-test runs:

- **Primary Metrics:**

- **P95 Latency (ms):** The response time that 95% of users experience, providing a better measure of perceived performance than the average.
- **Throughput (req/sec):** The total number of successful requests the system handles per second.
- **Error Rate (%):** The percentage of total requests that result in an error.
- **Time to Recovery (s):** The amount of time takes to restore a system, service, or component to full functionality after a failure or incident.

- **Secondary Metrics:**

- CPU and Memory utilization (%).
- Dropped Iteration (from k6).
- Trace durations and errors.

- **Analysis Method:** The primary analysis technique will be a comparative analysis of the pre-test and post-test data, calculating the percentage improvement for each key metric to validate the effectiveness of the implemented optimizations.

The percentage change for each KPI will be calculated using the formula:

$$\text{Improvement \%} = \left(\frac{\text{Pre-Test Value} - \text{Post-Test Value}}{\text{Pre-Test Value}} \right) \times 100$$

This quantitative comparison will serve to either support or refute the research hypothesis (H_1) and provide a concrete answer to the research questions.

3.5 System Environment

The research is conducted in a controlled cloud staging environment designed to closely replicate the production setup:

1. **Infrastructure:** The system is hosted on Private Virtual Server (VPS) from Contabo and IdCloudHost:

- **Monolithic ERP**

- Cloud provider: Contabo
- Server region: Singapore
- Operating System: Ubuntu 22.04
- Processor: Generic x64 8 vCPU
- RAM: 30 GB
- Storage: 400 GB

- **Microservices**

- Cloud provider: IdCloudHost
- Server region: Jakarta, Indonesia
- Operating System: Ubuntu 20.04
- Processor: Generic x64 2 vCPU
- RAM: 4 GB
- Storage: 160 GB

2. Software Components:

- **Monolithic ERP:**

- Application: Odoo Community Edition v17.0
- Host OS: Ubuntu 22.04.2 LTS.
- Database: PostgreSQL v14.

- **Microservices:**

- Framework: NestJs (TypeScript)
- Containerization: Docker.
- Database: PostgreSQL v14, MySQL v9, MongoDB v15, Redis.

3.6 Libraries and Tools

3.6.1 *Prometheus*

Prometheus is an open-source monitoring and alerting toolkit originally developed at SoundCloud and now a part of the Cloud Native Computing Foundation (CNCF). It functions by collecting and storing metrics as time-series data, allowing users to track the performance and health of your systems and applications. Prometheus utilizes a pull model, where it periodically scrapes metrics from configured endpoints, and features a powerful query language called PromQL for analyzing this data. This enables flexible and detailed monitoring, the creation of insightful dashboards (often in conjunction with tools like Grafana), and the configuration of alerts to be notified of potential issues. Its dimensional data model, service discovery capabilities, and strong integration with container orchestration systems like Kubernetes have made it a cornerstone of modern, cloud-native observability.

3.6.2 *Jaeger*

Jaeger is an open-source, end-to-end distributed tracing system that enables developers to monitor and troubleshoot complex, microservices-based architectures. Originally developed by Uber and now a graduated project of the CNCF, Jaeger provides visibility into the entire lifecycle of a request as it propagates through multiple services. By capturing and visualizing this journey as a "trace," which is composed

of individual "spans," it allows for detailed performance and latency optimization, root cause analysis of errors, and understanding of service dependencies. Jaeger's architecture, which includes components for collecting, storing, and visualizing trace data, along with its compatibility with open standards like OpenTelemetry, has made it a fundamental tool for achieving observability in modern, cloud-native environments.

3.6.3 *ELK Stack*

The ELK Stack is a powerful, open-source platform designed for searching, analyzing, and visualizing log data in real time, and is a popular solution for log management and analytics. It consists of three integrated components: Elasticsearch, Logstash, and Kibana. Elasticsearch is a distributed search and analytics engine at the heart of the stack, responsible for storing, indexing, and providing fast search capabilities for large volumes of data. Logstash is the data processing pipeline that ingests data from a multitude of sources simultaneously, transforms and enriches it, and then sends it to a "stash," like Elasticsearch. Finally, Kibana serves as the visualization layer, providing a user-friendly web interface to explore, visualize, and create interactive dashboards and charts from the data stored in Elasticsearch. Together, these three tools form a comprehensive solution for monitoring infrastructure, troubleshooting applications, and analyzing security events by providing a centralized logging environment.

3.6.4 *Grafana*

Grafana is an open-source interactive data visualization platform that allows you to query, visualize, alert on, and understand your metrics no matter where they are stored. It is widely used to create and share dynamic dashboards that provide insights into the performance and health of systems, applications, and infrastructure. Grafana connects to a multitude of data sources, such as time-series databases like Prometheus and InfluxDB, relational databases like MySQL and PostgreSQL, and various cloud services. Its strength lies in transforming complex data into beautiful and easy-to-understand graphs, charts, and alerts, making it a cornerstone tool in the fields of observability, monitoring, and data analysis, often used by DevOps teams and system administrators.

3.6.5 K6

k6 is a powerful, open-source load testing tool designed with a "developer-first" mindset for testing the performance and reliability of APIs, microservices, and websites. Acquired by Grafana Labs, k6 has gained significant popularity for its ease of use, high performance, and seamless integration into modern software development workflows. Unlike traditional, often complex and resource-heavy testing tools, k6 is built to be a flexible, command-line driven utility that empowers developers and QA engineers to write and execute performance tests as a natural part of the development lifecycle.

At its core, k6 works by simulating virtual users (VUs) that execute test scripts written in JavaScript. This choice of scripting language lowers the barrier to entry for developers, allowing them to leverage familiar syntax and a robust ecosystem of JavaScript libraries to create sophisticated and realistic test scenarios. The tool itself is written in Go, which ensures high performance and efficient use of machine resources, enabling the generation of significant load from a single machine. Tests are configured and initiated via the command line, making it straightforward to automate and integrate into CI/CD pipelines using tools like GitHub Actions, Jenkins, or GitLab CI.