National College of Ireland

Enhancing Microservice Performance: A Hybrid Model Combining Service Discovery and Circuit Breaker Patterns in Microservice Deployments

MSc Research Project
Cloud Computing

Ritika
Student ID : 22208691

School of Computing
National College of Ireland

Supervisor: Diego Lugones

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Ritika
Student ID:	22208691
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Diego Lugones
Submission Due Date:	12/08/2023
Project Title:	Enhancing Microservice Efficiency: A Hybrid Model Combining Service Discovery and Circuit Breaker Patterns for Optimized Performance
Word Count:	7301
Page Count:	22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	11th August 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Microservice Performance: A Hybrid Model Combining Service Discovery and Circuit Breaker Patterns in Microservice Deployments

Ritika
22208691

Abstract

Microservices are a medium to contribute and build a dynamic, high functioning application that can be used worldwide by leveraging cloud services. The requirements can be easily integrated under different frameworks and the configurations that come with them. The main question that arises is what happens when any service of the application faces any issue or downtime. The application should be made flexible and scalable while keeping the factor of high availability in mind. With many organizations migrating to cloud-based infrastructures, it becomes complex to manage the response time and latency of the independent microservices. For instance, Amazon faces costs of 1% sales for every 100ms of delay **van Vessum (2024)**. Therefore, enhancing the responsiveness of microservice dynamically, this paper explores two widely used design patterns: Service Discovery and Circuit Breaker. We conducted experiments using both patterns individually as well as a combined architecture on backed services developed using the Spring Boot Framework. Under the combination, it is observed that the Circuit Breaker pattern's fault tolerance and resilience while service discovery's efficiency for load balancing and dynamic routing brings together high performance architecture. The findings observed through experiments say that each pattern improves latency independently, while we proposed a hybrid model that significantly improves the performance by fourfold by harnessing the strengths of both patterns. This improvement contributes to the performance achieved that addresses the real issue of microservice latency, improving response time and making application fail safe.

1 Introduction

The microservices architecture divides the application into smaller, independent services that interact with each other via a network. This approach distributes the application, making it more scalable and resilient. However, because of the large number of service- to-service interactions, managing traffic between these services can be difficult. Efficient traffic management is critical to ensuring performance and reliability, so service discovery and registration are required components.

Microservice interaction response times are under tremendous pressure from the increasing demand for immediate and seamless user experiences. Users may become irritated with even small delays, which could cost businesses money. Latency becomes a major concern when the volume of inter-service communication demands rises, particularly in

virtualized environments. Microservices frameworks replaced tightly coupled monolithic designs commonly used in cloud computing. The smaller and independently deployable APIs and services increase scalability and flexibility. Because of its loose coupling, it also allows for fast modifications. Nonetheless, handling thousands of microservices degrades user experience by adding to complexity, latency, and network traffic.**Shadija et al. (2017)** shows how distributed systems can experience noticeable communication lags between microservices, which can cause a domino effect of performance problems. Systems that have good response times are better able to handle heavy traffic loads, remain responsive, and maintain service-level agreements (SLAs).
Having expertise in new insights, practices, and principles of microservice architecture is necessary to design successful microservice-based applications. However, there is still much to learn about these aspects. One such element is the application of patterns; as experts come across and resolve issues brought about by this architectural approach, common answers to issues start to surface. By providing software architects with a variety of approaches to common problems, the documentation of these patterns lowers the technical risk associated with their projects by avoiding the need to use novel and unproven designs**Vale et al. (2022)**. Microservices offer increased agility, scalability, maintainability, and performance; however, to fully realize these promises, one must have a thorough understanding of the underlying principles. Even though this architecture has been the subject of much hype in recent years due to the support of major industry players, the best ways to implement it are still being researched.

1.1 Motivation for identifying the Middleware Layer

Building on the previously established concerns about microservices and their interactions complexity, the distributed nature of such services can present a significant challenge: high latency. This is due to increased network traffic caused by constant communication between services. Current approaches to addressing this issue frequently involve overloading resources, which leads to inefficiencies and higher costs **Desina (2023)**.
However, the most efficient solution to optimize the middleware architecture is to focus on this layer. We may ensure effective communication within services along with optimized resource management. This reduces latency, costs, and overall system responsiveness. This study aims to investigate how middleware design patterns can be used to achieve these objectives.

1.2 Research Objectives:

This study seeks to investigate the most recent advances in microservice design and architectural patterns. It focuses on developing a framework that combines the Circuit Breaker and Service Discovery patterns with load balancing capabilities. The next step is to implement and simulate these designs in real time across a variety of cloud microservices scenarios. Considering these scenarios, the project will evaluate the performance, latency, and resource usage of the implemented patterns in a cloud environment, which brings us to our research question :
How do Circuit Breaker and Service Discovery patterns perform in combined implementation with simulating workload across diverse use cases of cloud

microservices?
This research also ensures industry applicability by considering specific use cases, regulatory compliance, cost effectiveness, seamless integration, consumer experience, DevOps integration, vendor-agnostic implementation, future-proofing strategies, and use case specificity are all considered to ensure industry applicability. The final step is to address how these patterns perform in real-time simulations across a variety of cloud microservices use cases.

1.3 Structure and Outline

The remaining sections in this research thesis are divided into five sections as mentioned below:

- Section 2 discusses the related works that have been applied to the subject of latency, fault-tolerance and response time in cloud microservices.
- Section 3 of the thesis holds the details regarding the methodology of the design patterns used.
- Section 4 discusses the design and implementation of the proposed architecture.
- Section 5 analyses the experiments and its results.
- Section 6 provides the discussion of the analysed experiments.
- Section 7 is last and final section to provide conclusion and future work.

2 Related Work

2.1 Industrial insights on Microservice Design Patterns

A study that actually investigates the effects of various microservice design patterns on quality attributes using semi-structured interviews with experts. Patterns like Circuit Breaker and Service Discovery are analysed as well as experimented for their impact on scalability, performance, and availability.

The results validate theoretical trade-offs and provide new insights, however, limitations include a small size of the sample based in Portugal and the qualitative aspect of the data, which might not fully represent industry best practices **Vale et al. (2022)**.

2.2 Managing Assurance and Technical Debt in Microservices

This study actually uses 17 semi-structured discussions with experts from ten firms to examine technical debt management and availability assurance in microservice architectures. It highlights procedures, technologies, and problems encountered in guaranteeing the availability of microservices; it covers architectural principles, code review, and tools for quality analysis in the CI/CD pipeline. The study notes that while there is an absence of architectural resources and metrics for evaluating the macro architecture of microservice systems, manually operated code review jobs cannot be fully automated.

Although the study highlights the requirement for instruments and statistics for macro-architecture assessment, generalizability is limited by the tiny, geographically confined samples and dependency on subjective perceptions. **Bogner et al. (2019)**.

2.3 Circuit Breaker Patterns in Cloud-Native Apps

The research findings of this study actually emphasize the importance of the circuit breaker pattern in cloud-native apps (CNAs) for preserving reliability in distributed systems throughout this study. The report evaluates various circuit breaker libraries as well as offers suggestions for improvements in the field. It was launched in the Global Conference on Information Technologies and Communications (GCITC) in 2023. Experimental verification with Internet of Things applications shows that circuit breakers actually increase availability and performance.

The study actually highlights the need for dynamic and adaptive circuit breaker techniques in a way to increase resilience and fault tolerance. Comparing different sets of libraries can actually provide developers as well as researchers with valuable insights that can guide future research and practical applications in a way to improve the performance and dependability of CNAs. **A and Sebastian (2023)**

2.4 Self-Healing MSA: Dynamic Version Management

In this research study dynamic version management is proposed using an innovative self-healing microservices architecture (MSA) that can adjust to service upgrades, infrastructure failures, and traffic spikes. Utilizing containerized microservices controlled by numerous VM nodes to ensure resilience, the method combines an updated version of the administrator with service discovery strategies. Through planned testing using chaos engineering in a production environment, the project aims to increase system efficiency and performance.

But the study mostly ignores unforeseen setbacks and dynamic scaling in favor of concentrating on a single company project and preliminary theoretical validation. In-depth validation and additional research are required to effectively represent the complexities of the real world. **Wang (2019)**.

2.5 Impact of Containers and Overlay Networks in Cloud Microservices

This paper actually analyses how overlay networks, encryption, and containers actually affect HTTP-based and REST-like services in Infrastructure-as-a-Service (IaaS) cloud environments. Overlay networks have the potential to reduce network performance by 30%-70%, whereas containers only have a 10%-20% impact. The actual impact of encryption is negligible, particularly for brief messages. Installing SDVN routers on multi-core, non-containerized virtual machines (VMs) can actually help minimize message sizes and address overlay network performance issues.

Now despite of these drawbacks, the flexibility as well as management that both, container as well as SDVN technologies provide for scalable distributed cloud platforms emphasize that how crucial it is to take performance issues into account when utilizing these technologies. **Gotin (2018)**.

2.6 Analysis of Performance Metrics of Microservices in Cloud IOT Environments

This paper actually presents a case study on using performance metrics for a threshold-based auto-scaler to manage message queues in Cloud IoT solutions, preventing over-loaded queues as well as SLA violations. It also evaluates metrics for scaling I/O-intensive as well as compute-intensive microservices, highlighting the superiority of message queue metrics over CPU utilization.

It has been demonstrated that message queue metrics-based thresholds are far more adaptable to variations in the microservice's properties. For this reason, in similar setups, we emphasize the advantages of depending on message queue metrics rather than microservices CPU utilization. **Gotin et al. (2018)**.

2.7 ServiceRank: Detecting Anomaly in Large-scale Microservice Infrastructures

In this work, the researchers actually present ServiceRank, an innovative framework for root cause analysis as well as anomaly detection in microservice architecture. ServiceRank has an RCA module to find suspected services without predetermined thresholds and a detector to track anomalies. It uses a second-order randomization movement based algorithm for root cause analysis and builds impact graphs using relationship extraction to look into particular anomalies.

ServiceRank improves anomaly diagnosis quickly and is applicable to a wide range of complex systems. Its efficiency in identifying anomalies as well as categorizing throughput and latency factors is demonstrated by experimental analysis. **Weilan et al. (2022)**.

2.8 Performance & Granularity : A comparitive study on Microservices

By contrasting two methods—deploying APIs within a single container and installing them across several containers—the researchers tested out microservice deployment. In both configurations, they calculated the round-trip time (RTT) for every API. It was observed that the RTT variance among the two approaches increased significantly as request bandwidth increased.

The RTT difference for almost 10 requests was more than 150 ms, and for 100 requests, it was more than 850 ms. This suggests that the single container strategy would have higher latency with higher load.**Shadija et al. (2017)**.

2.9 Real-time Auto-scaling of Cloud Microservices

An autonomous autoscaling procedure for microservices in cloud-based systems with Quality of Service (QoS) limitations is presented in this research. Two parts make up the process: reinforcement learning agents that establish autoscaling limits based on resource requirements and quality of service (QoS) and a standard autoscaling technique in Kubernetes (GKE) that adjusts to application-specific resource needs.

The results of the experiment indicate improved efficiency and performance with a 20% increase in microservice response time over the default autoscaling paradigm. **Khaleq and Ra (2021)**.

2.10 Integration of Enterprise Patterns in Microservices

This white paper offered a method for designing microservices that actually demonstrates a model-oriented as well as a pattern-based approach. This method also provides a smooth transition from the working and architectural model methods to operational deployments. Additionally, the researchers demonstrated the implementation of their suggested approach on a working example using OpenFaaS, which is used for logic governance through integration, and Kubernetes, which is utilized as a system for container orchestration. This served to validate the approach's viability.

While the researchers did show that the suggested approach was feasible by putting it into practice through a case study, they were actually unable to assess the effectiveness of their research because they neglected to track the response times of the services they validated and put into practice **Černý (2018)**.

2.11 Cloud-based Persistence System: Quantifying the Response Time Latency of Microservices

This study proposes a statistical performance model that predicts the percentage of requests needed to meet service level agreements (SLAs) for an event-driven cloud object storage system. The main issues with interleaving multiple disk processes on storage servers are addressed by this concept. It also measures the effect of the acceptance waiting period on the system's response latency. In order to conduct experiments, they configured one process for each storage device in one scenario and sixteen processes for each storage device in another. Next, the SLA percentage metrics are used to compare these scenarios.**Su et al. (2017)**

Based on the aforementioned experiments, their suggested model outperforms the baseline models in terms of response latency prediction errors, with a reduction of nearly 73%.

2.12 Adaptable Microservices patterns with potential Futuristic Utilization

The researchers' goal in this work was to conduct a thorough analysis of design patterns and how they impact microservice-based systems' extensibility. They took into account three distinct extensibility approaches Microservice Internal Adaptability, Extensibility

of Microservice, and Extensibility of the System when conducting this experimentation. Based on six distinct domains: communication, data, query, decomposition, deployment, and interface, they nearly tested 18 different patterns of system design.

During analysis, they found that most patterns had a positive impact on extensibility; however, this investigation did not include inner executions of any of the services, which would have provided more reliable results to support these implications **Daniel et al. (2024)**.

2.13 Distribution of Microservices based on Cloud Design Pattern

This research that we are discussing about actually demonstrates two different distribution strategies, which are the random distribution as well as design pattern-based distribution, with an emphasis on container-based microservices methods based on pre-distribution. The microservices are dispersed randomly among the available data centers for random distribution. On the other hand, the pattern distribution containerizes the microservices. Their testing demonstrated that, when compared to the random distribution strategy, the design pattern-based distribution actually produced better results and consequently required less response time for the cloud computing environment.

When a distribution based on design patterns was used in place of a random distribution, the study actually demonstrates progressive improvements. Additionally, their research has possible potential for extension, which implies that a more empirical in- vestigation into better software architecture patterns is anticipated in a way to increase this time latency more successfully and potentially yield better outcomes **Saboor et al. (2021)**.

2.14 Contribution of the Study

When it comes to the contribution, the most important question that arises while discussing cloud-based applications is how scalable they would be, whether they can actually handle large payloads or high request volumes, as well as what shall happen in the event of a failure and how the service will handle it.

The literature reviews a variety of approaches, techniques, assessments, as well as experiments with the shared objective of managing fail-safe scenarios and the response time latency as discussed above. But those existing research lacks comprehensive analysis mainly in how both the patterns perform when combined together under different conditions.

As a result, while delving into our own research, we discovered that there has been relatively little research on the topic of microservice design patterns, yet not fully explored for the needs of integrated approach that enhances the dynamic nature of microservices. This actually allows us to properly position patterns based on the characteristics of microservices even before the services are called. This research addresses the gap by comparison and integration of different patterns as well as effectively integrates in a way to enhance the performance of the microservices while reducing the response time delay.

3 Methodology

As seen in the preceding section, several literary works have been discovered to alter a range of parameters in order to enhance the latency of a microservice's response time. Accordingly, on the basis of related experimental research, several of these efforts have been determined to increase performance and subsequently reduce latency up to a certain factor. I want to take a close look at the function of these two design patterns integrated within microservice communication and how we can manage the fallback mechanism. Service-to-service communications are the lifeline of modern microservice architectures. However, when traffic increases, it becomes difficult to manage communication between services. These high traffic can strain an individual service thereby increasing failures. Therefore, handling a high volume of traffic between these services can become extremely complicated. When a single service is strained by heavy-load and high volume of requests, the communication weakens, and that leads to failure of a single service which may have a cascading effect on the entire system. Hence, to minimize this risk, a circuit breaker pattern is required in between services.

The circuit breaker is a design pattern that allows the detection of faults and captures the logic of preventing a failure from recurring while performing maintenance, addressing unforeseen system, or managing a temporary failure of the external system. Working much like an electrical circuit breaker, it trips the circuit when a defect is detected, preventing further damage and enabling some backup processes.

Service registration and service discovery automate the process of finding and connecting microservices to each other. We can say that the addresses will not be hardcoded. In this way, the addition or removal of services will not have any effect on the whole system**Khan (2024)**.

Implementation of standardized tools and frameworks in a service-oriented architecture can go a long way toward handling a range of concerns. Among which are metrics, exception tracking, logging, distributed tracing, health checks, configuration, security, service discovery, and circuit breakers. The following can help with integration: taking advantage of and using service meshes at the network level to solve a variant of problems, other tools like Spring Boot, ELK Stack, Eureka, and resilience4j. A microservice template is a means of 'self-decoupling' these tasks from the application code and allowing uniformity in the setup of new services, reducing its complexity. You are not meant to write code to set them up from scratch every time you want to add a new service. Building any service on top of the micro-service chassis, you can thrive and create your services at a much faster rate. A microservice chassis offers a good starting point which provides a typical framework including necessary tools, libraries, and configuration. This depends on the configuration targeted at various environments, which should be treated separately for services meeting all requirements such as network locations of databases, external endpoints of message queues, and corresponding credentials to an extent by centralized configuration management tools like Eureka, Spring Cloud Config, Manifests or Kubernetes ConfigMaps and Secrets. This ensures uniformity, reduces setup overhead, and increases the development speed of new services.

By implementing these components in practical scenarios, we observed positive impact for Service Discovery which includes dynamic routing and load balancing. This shows that new instances can register easily without any manual configuration. Also, the mechanism performs updates for the registry when needed in order to scale up or down for the services. This when combined with fault tolerance of Circuit Breaker also scales while preventing overload on the system in accordance with managing different loads with

reliability.

The above architecture illustrates a combination of 3 independent, loosely coupled service interacting with each other under the Spring Boot Framework. To set the context, Service discovery design pattern Eureka is implemented with services and one service is integrated with circuit breaker pattern. The Spring Admin dashboard provides the

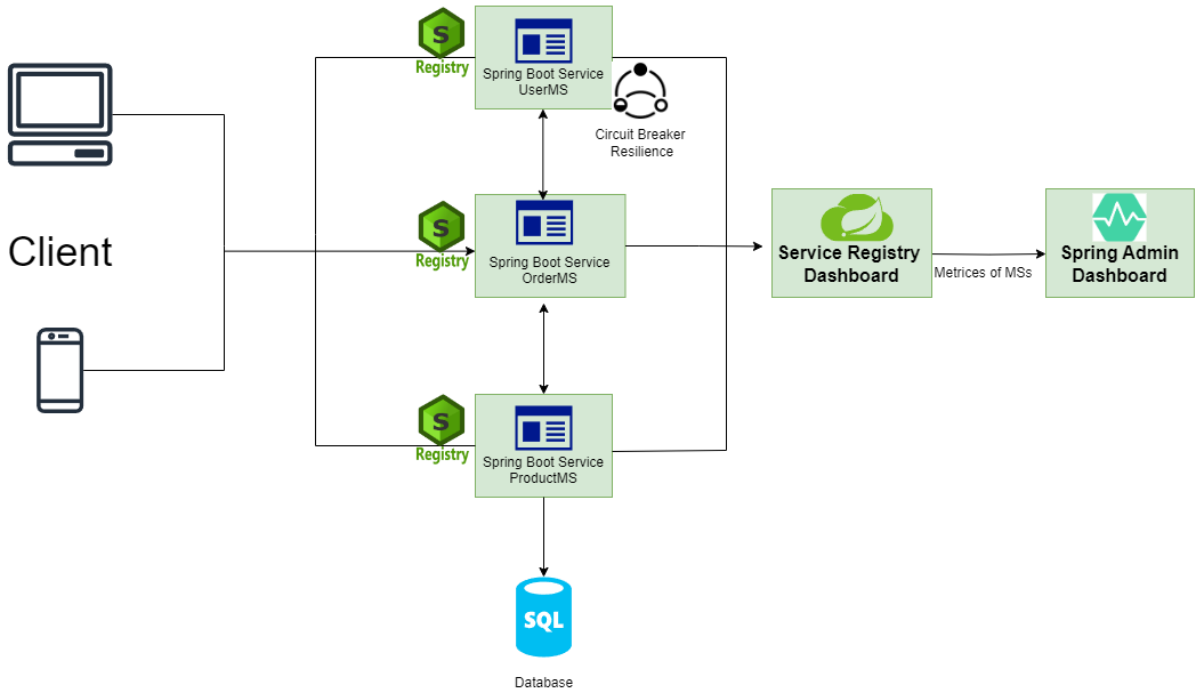

Figure 1: Microservices Dynamic Functioning Using Design Pattern

evaluation grounds where performance, logging, and other metrics can be visualized. This diagram **Figure 1** illustrates a microservice architecture implemented using Spring Boot, highlighting the role of service discovery and monitoring dashboards. In the context of research, which compares the performance of Circuit Breaker and Service Discovery patterns in real-time simulations across diverse use cases of cloud microservices, this diagram provides a practical example of how both patterns are integrated within the microservice ecosystem.

The communication between the services are tested under different scenarios so that we can observe the peak throughput of requests and how the response time gets affected. The different conditions are as follows:

- CASE 1 - Simple GET request; NORMAL OPERATION
- CASE 2 - POST request; INCREASED LOAD
- CASE 3 - POST request with heavy payload; HIGH LOAD

The operations are handled with RESTful APIs with CASE 1 as a GET request to the application that is processed and fetched from MySql. Secondly, POST request with payload is processed with business logic and through MySql response is returned. Lastly, a payload with attached document as a POST request is sent for processing to store in MySql. These conditions are processed within three microservices as shown in **Figure 1**. These conditions are used to obtain the proper outcome for evaluation. The Spring Boot Admin, an open-source web interface project is a plug-in which we have used to manage, monitor and look into the insights of the whole application. The client services are registered with the Spring Boot Admin server so that we can use to look at the metrics needed for research Aibin (2024).

3.1 Research Procedure

This section gives a detailed explanation of response time delay of microservices under both architecture design patterns. This will lead to the discussion of comparative analysis. In addition, the combination of both models is also presented in order to analyze a better way to improve the issue of latency.

3.1.1 Service Discovery Design Pattern Model

We are aware that microservice-based applications exhibit dynamic behavior. For example, if you call an available service in any instance, it will be called by its IP address and port number. However, due to any circumstance, if the service goes down, perhaps due to network unavailability, poor resource management, or planned service maintenance and becomes available again after a period of time, it dynamically acquires a new IP address and port number. So consequently, if a service is called by its designated IP address, it becomes quite complicated to maintain the same IP throughout its availability. However, there are cloud platforms that provide services to maintain designated IPs of any microservice, as an instance, Amazon Web Service (AWS) does provide static IPs, but it is indeed a costly operation to assign a static IP individually to every microservice and maintaining the availability of the microservice.

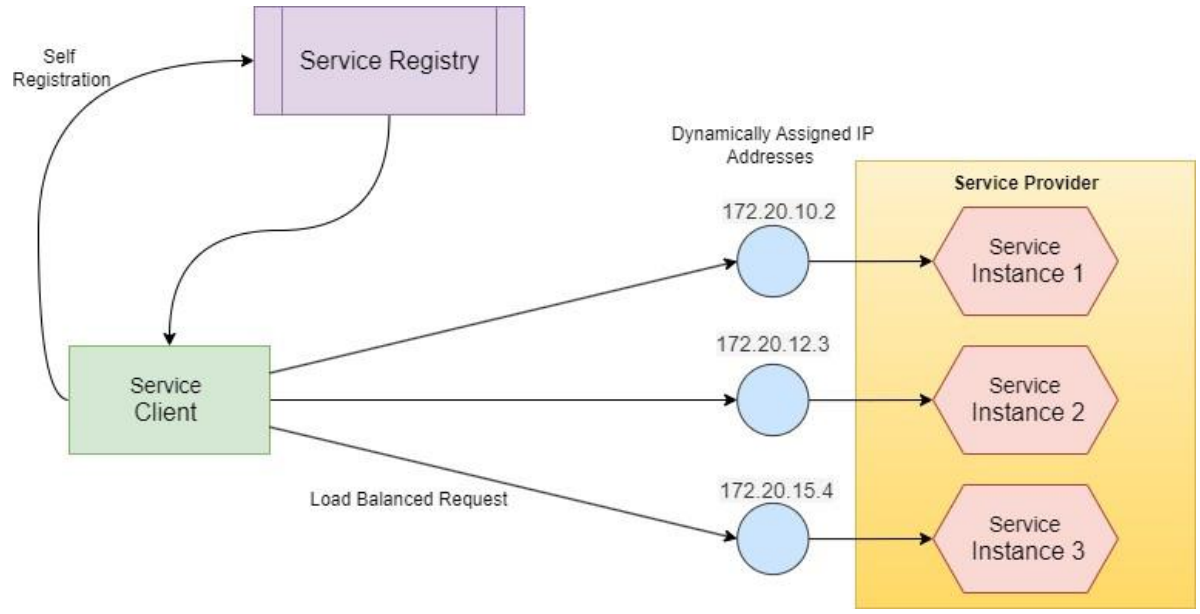

Figure 2: Service Discovery Working Model

This is where the need of service discovery design pattern becomes ideal and relevant. The client gets advantages by knowing the name of a service instance. The service discovery pattern follows a simple architecture of calling a service by its name rather that its IP by maintaining a service mesh/registry which registers the name of every microservice which are available at an instance of time and these services get registered by their respective registry client. So when a service faces a downtime, it gets removed from the registry and when it becomes available, the registry client associated with it registers its name again to the registry.

Figure 2 elaborates the working of registered services in service registry via a registry server. The server configuration is responsible for the self-registration process of the ser-

vices. The service instance invokes the registry service to instantiate the API registration to the Register. This also includes the health check URL which, after monitoring the API is able to handle the requests. The registry is also responsible to invoke a request to API to prevent the expiration of registration Awad (2024) Bhojwani (2022).

3.1.2 Circuit Breaker Design Pattern Model

Organizations face a high volume of requests to services so when any service is down or is under maintenance it might lead to whole application coming down. Managing fault tolerance and resilience in microservices becomes of great importance as if not managed leads to delay in response time and overloads the whole architecture. This drawback can be handled by introducing Circuit Breaker design pattern with microservices. The system can be made to recover from the issues and cease cascading failures. Here we understand with the flow where requests travel from service 1 through service 2, service 3 and service 4 to perform certain task. The **Figure 3** below shows that there is some issue with service 4 so the service starts to slow down. The system repeatedly tries to execute an operation which fails repeatedly. The requests going to service 4 starts queuing up.

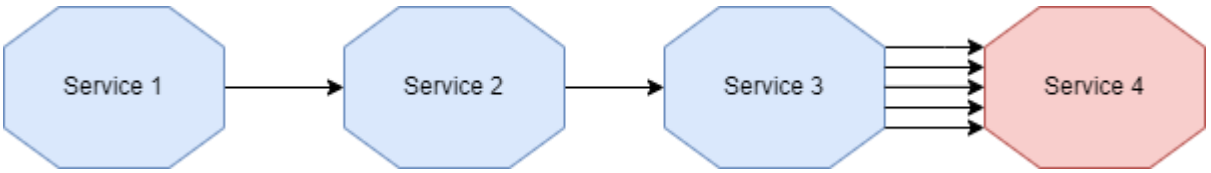

Figure 3: Unavailability of Service 4

Here, we observe the latency between the services. Because of this latency, the requests starts queuing up in all related services which leads to back-propagating the delay as shown in **Figure 4**.

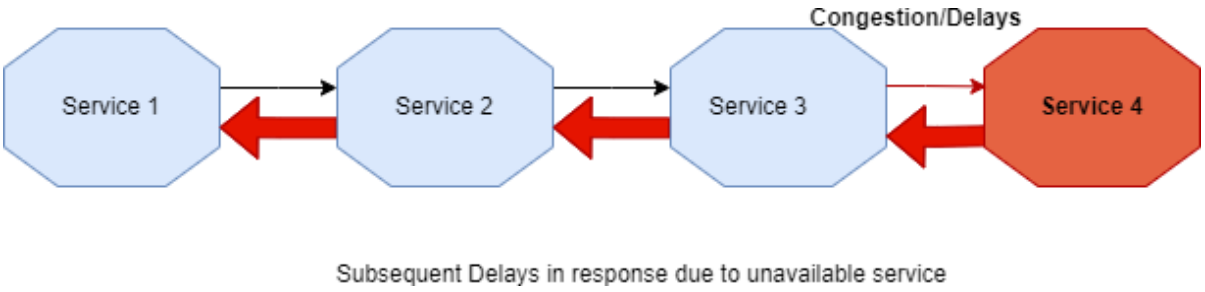

Figure 4: Unavailability of service 4 leads to downtime of whole application

To resolve this, the better approach would be that if a particular service is taking more time than usual, then don't send any more requests to that service. This prevents an increase in slowing of other services. In microservice communication, when the number of errors in a given time frame is beyond an acceptable limit, the circuit opens, thereby preventing further flow and protecting other parts of the application, as shown in **Figure 5**.

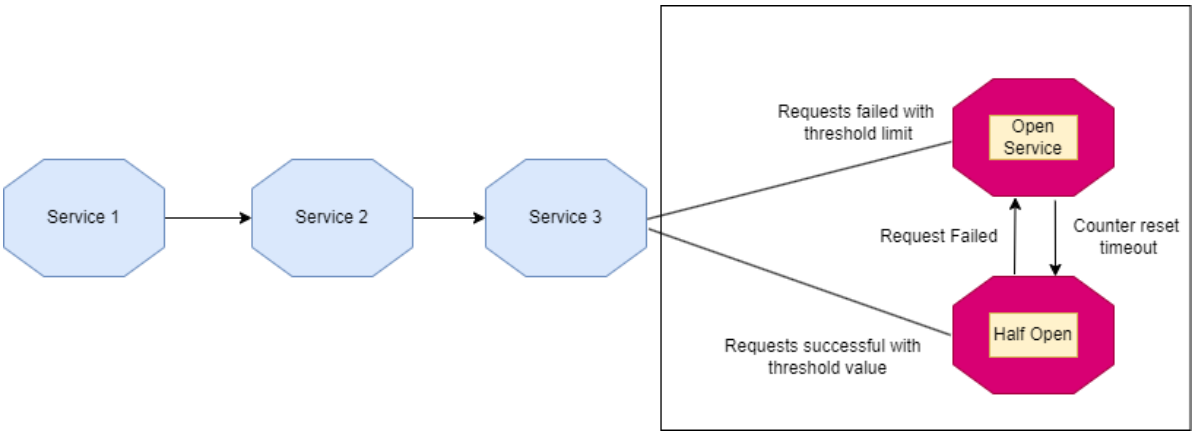

Figure 5: Circuit Breaker applied to service 4

3.1.3 Combination of Service Discovery and Circuit Breaker Design Pattern

In above sections we have noticed the two patterns focuses on resolving different aspects of issues faced in microservices, so we present here an approach to combine the two patterns together to overcome the failures and latencies.
This approach comprises three microservices registered with a service registry like Netflix Eureka which comes with Spring Boot configuration. This enables dynamic decoupling of endpoints of the services and also sums up to load balancing and horizontal scaling.
In order to address any failure at any point of time, we have implemented circuit breaker for Service 2 which receives calls from Service 1. By the help of circuit breaker, we can monitor call success and failures, therefore avoiding cascading failures as discussed in **Figure 4**. Now whenever the service is down, a fallback mechanism comes into play by either not hitting the service again and again or reverting the requests to alternate services. The circuit is also responsible to check whether Service 2 has recovered on regular intervals and if its healthy the communications goes back normal.
The architecture acquires resilience, efficiency and least latency by integration of both the patterns. The circuit breaker provides fault tolerance and graceful degradation, while service discovery ensures dynamic load balancing, better management, and monitoring of registered services. This ecosystem is capable of maintaining high availability with scalable performance.

4 Design Specification & Implementation

Designing the application based on their use case and functionality is critical in building the foundation. Logic-based microservices that communicate with each other with real-time data. Following the implementation process, we are comparing the real-time performance of microservices implemented under two different architecture models: Service Discovery and Circuit Breaker. The application functionality is the same for both models which are processed under three conditions, namely: Normal operation, Increased load, and High load. The application is developed in Java language using the Spring Boot Framework. The performance metric is **Time Latency** which is being captured from sending the service request to the services, and is processed between them and sending the response back to the consumer/client.

4.1 Service Discovery Model Based Design

The implementation process for designing Service Discovery Model includes the base application including three microservices handing different business logic. The front-end which is developed in ReactJS for user interface while for the back-end the development is on Java language under Spring Boot Framework is used. The application is developed to receive HTTP requests with GET and POST operations. In a way to achieve the implementation of the Thread Management based model, we shall be using the technical stack, which involves an application in front-end for user interface developed in ReactJS. Talking of the back-end part, we have Spring Boot Framework exclusively developed in Java where the business logic for the endpoints, that is, POST as well as the FETCH operation will be developed as shown in **Figure 6**. After the proper functioning and relevant unit testing of end points, they will be exposed to the front-end application for utilization or may be for a group of end users to use them in accordance to their requirement usage.

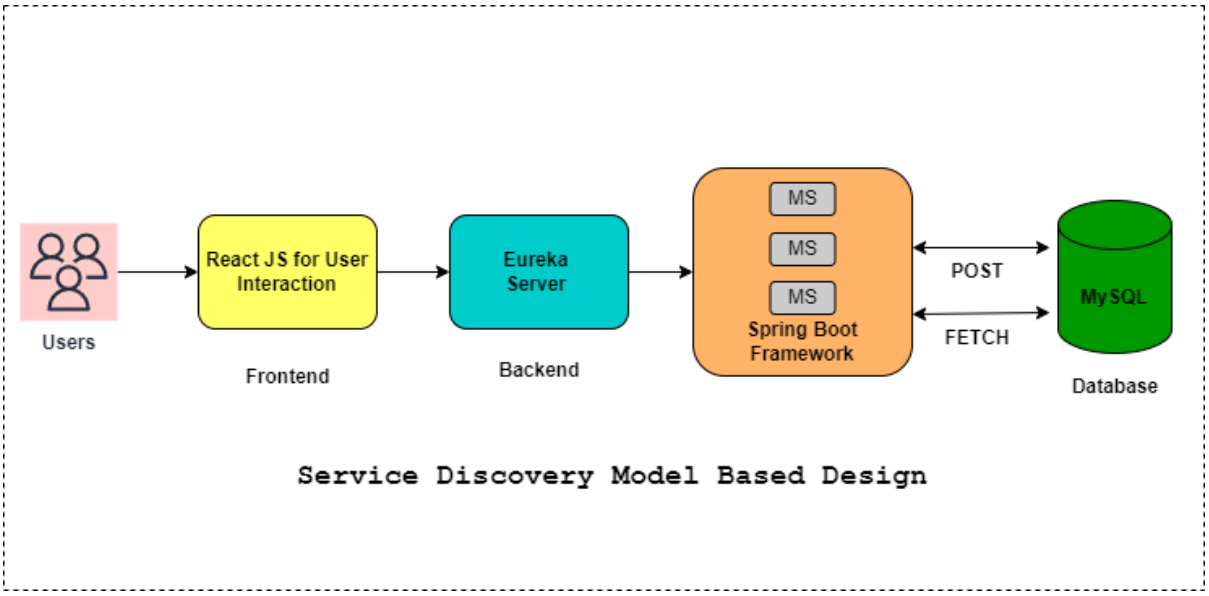

Figure 6: Service Discovery Model Based Design

4.2 Circuit Breaker Model Based Design

For the implementation of Circuit Breaker model we follow the same architecture design as mentioned above. The services here are not registered on the eureka service registry; instead, we implement a circuit breaker mechanism in service A which calls service B holding the business logic to retrieve a response with some payload. If service B is down and does not respond, the circuit breaker in service A trips when it touches a failure threshold which prevents further calls to that service avoiding cascading failures. The service A is also configured with a Fallback mechanism where a default request is made to alternate service or to the same service in some time configured with the same as shown in **Figure 7**. It retries the down service in regular intervals to resume the normal operation based on different configurations of Fallback mechanism, thereby reducing the latency and ensuring smooth processing of queries and isolating the failures.

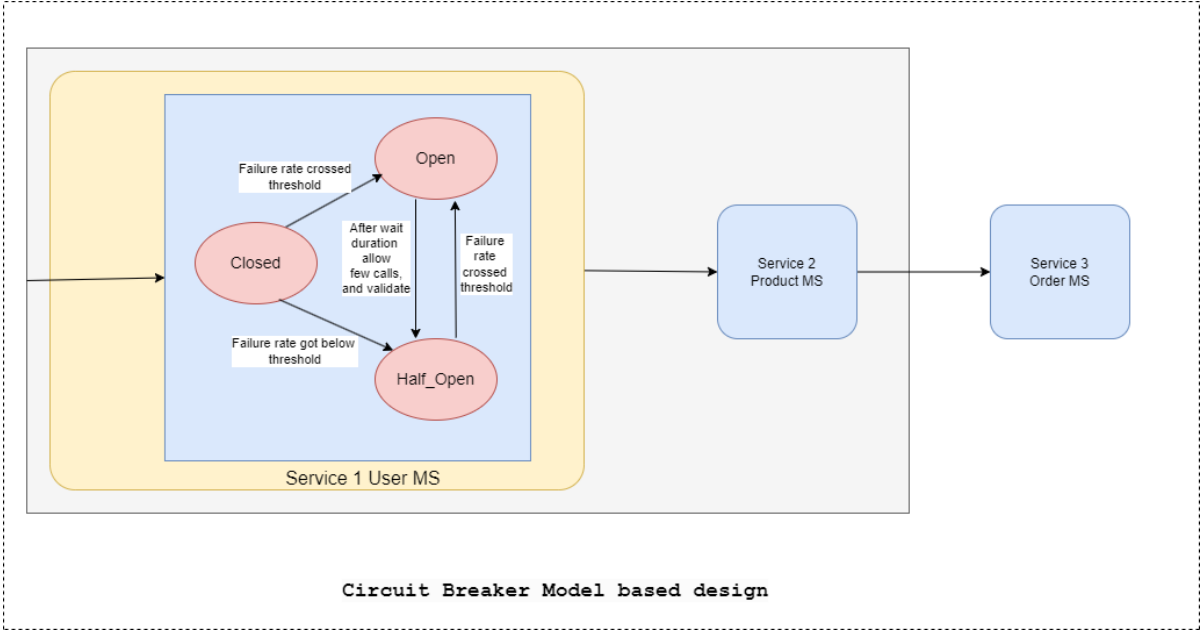

Figure 7: Circuit Breaker Model Based Design

4.3 Combination

In this section, the above configurations are made together in those three microservices. We register the microservices with service registry, thereby dynamically allotted IPs to the services. Eureka helps service 1 to find available instances of other services. The service A is integrated with circuit breaker pattern with fallback mechanism which includes factors like minimumNumberOfCalls, failureRateThreshold, waitDurationInOpenState and permittedNumberOfCallsInHalfOpenState. These attributes configure the circuit breaker's sensitivity and recovery behaviour making sure after the circuit trips it comes back to normal operation. These implementations are performed on three independent microservices, each performing different tasks as shown in **Figure 8**. The Spring Boot framework provides an ease to set up the two patterns so as to manage and monitor the services through Spring Boot Admin.

5 Evaluation

Based on performance metrics collected from both models after implementing in their particular techniques as discussed in the prior section. In order to compare them, both models are examined in the same category of microservices with the same payload. The workload comprises different HTTP requests like GET and POST with different payloads, including high-load requests made to services. Services were made to stop functioning while creating downtimes intentionally to understand how the circuit breaker responds to the service unavailability. In this experiment a total of 9 experiments were done to evaluate the outcome, performance of the microservices. Each experiment included 3 times each with different conditions with Circuit Breaker pattern, Service Discovery pattern and Hybrid (combining both the patterns).

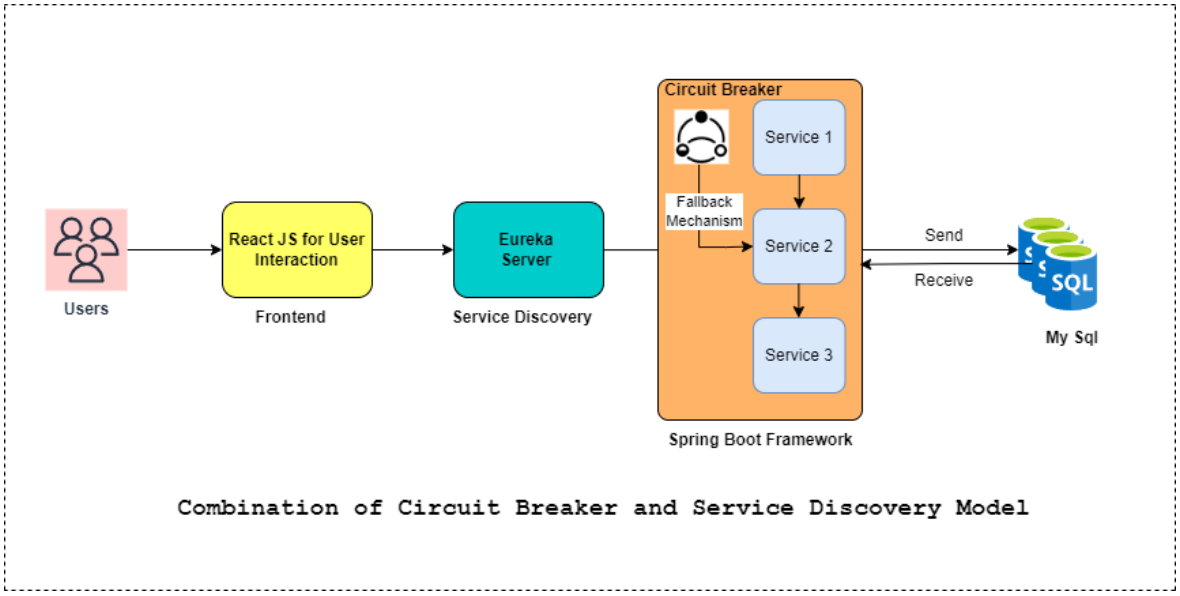

Figure 8: Combination of Circuit Breaker and Service Discovery

5.1 Evaluation of Circuit Breaker Pattern

The experiment was conducted to evaluate the performance, latency, and reliability of the circuit breaker pattern in cloud-based microservices. The main metrics observed while running the microservices through HTTP requests are total time taken to process (TOTAL TIME SPENT), and maximum time taken to response (MAX TIME SPENT). The application is also processed under different operations, namely Normal, Increased Load, and High Load. Normal operation is a simple Get method call with simple fetch operation, whereas under Increased Load we make POST call to services with some payload, and under High Load we send some document/image with payload so as to collect metrics under different conditions. These metrics provide the high level view of the system's ability to handle load, manage requests and response under different situations. The following observations are of services before and after downtime scenarios:

The evaluation spectrum focuses on Latency metrics on basis of comparison of same

Condition	Metric	Before Downtime	After Downtime
Normal Operation	Total Time Spent	12.5018 seconds	15.2512 seconds
	Max Time Spent	0.3299 seconds	0.0624 seconds
Increased Load	Total Time Spent	18.7234 seconds	21.2344 seconds
	Max Time Spent	0.4567 seconds	0.1345 seconds
High Load	Total Time Spent	25.6567 seconds	32.4991 seconds
	Max Time Spent	0.6789 seconds	0.8745 seconds

Figure 9: Metrics for Circuit Breaker Pattern

microservice architecture under two different design patterns. Keeping in mind, the requests send to the services.

5.2 Evaluation of Service Discovery Pattern

This section holds similar configuration set up as mentioned above. The experiments are performed on the same set of microservices. Based on that, requests are sent to particular microservices. Those requests are recorded and processed through the services and its interconnected functionalities, and with response we can record the response time, so as to increase the performance and to negate latency. The HTTP requests are considered with the same other attributes. The following observations are of services registered with Eureka Server (Service Discovery): Spring Boot Framework is widely used Java

Condition	Total Time Spent (seconds)	Max Time Spent (seconds)
Normal Operation	4.7212	0.7138
Increased Load	6.2345	1.1234
High Load	8.5678	1.5678

Figure 10: Metrics for Service Discovery Pattern

Framework and is standalone production ready spring grade set up. The configurations are easy, which allows features like metrics, external arrangements, and health checks which helps to explore the microservices to greater extent. The middleware layer in this framework has a lightweight HTTP routing system that connects to the framework with simple dependencies and annotations.
In this also we will keep Latency as main focus of performance metrics for evaluation and comparison. The base idea is the response time observed from the microservice through the middleware layer and service discovery pattern.

6 Discussion

From the above section, we have obtained the proper metrics in tabular form of the two patterns. The details are observed from the HTTP POST calls made to the microservices with JSON payload. The Spring Admin is an attached configuration of the Spring Boot Framework which helps to get the performance numbers and the health of the microservices to which requests are made. The services are internally called to perform certain operations and return with a response in JSON format (200/201 response code). Therefore, the time taken to perform certain operation and return back with a response is the major factor that is being considered on both the models.

6.1 Circuit Breaker Readings

From the above section, readings are taken from both the models and presented. Now, those readings should be plotted in a line graph to have a better understanding of the

whole model. The above graph allows us to analyse the latency metrics of microservice

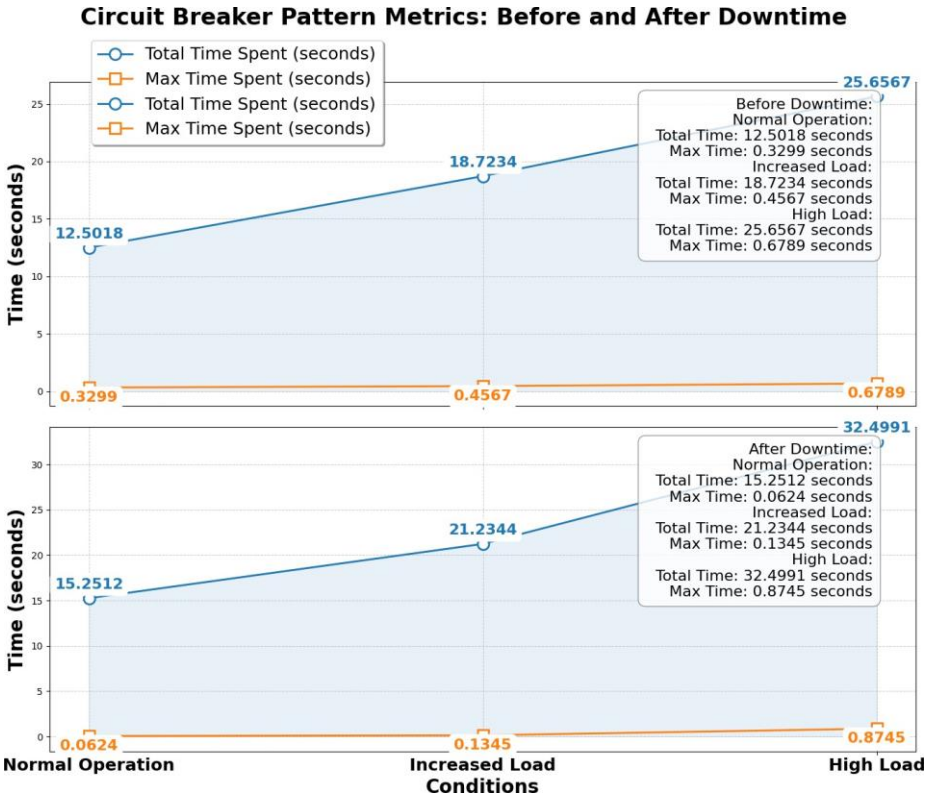

Figure 11: Visual representation of Circuit Breaker Pattern on test readings

architecture under Circuit Breaker pattern before downtime of service and during downtime of the service. The top graph shows **"Before downtime"** and lower one is **"After downtime"**. The x-axis shows conditions like **Normal Operation, Increased Load and High Load** while the y-axis represents **time** in seconds. Based on the tabular data we observed **Total Time Spent** (Blue line) and **Max Time Spent** (Orange line) are the two key metrics provided by the Spring Admin, Obregon (2023) a feature of Spring Boot Framework that allows to check the actuator health conditions of the services registered. The summarized blue line indicates the cumulative time taken to process all requests, whereas the orange summarized line represents the maximum time taken to process a single request.

The **Total Time Spent** is **12.5018 seconds** before the service downtime when the whole system works smoothly, which increases to **15.2512 seconds** during the service downtime under normal operations. But we can observe that under increased load and high load the graph sequentially increases. This shows that the spike could have been larger if the complexity of dealing with retries and failures were not used. The fallback mechanism in the system returns faster with the response of the service being not available at that moment instead of user keeping hitting requests and increasing overhead.

6.2 Service Discovery Readings

Now lets plot the observed readings for service discovery pattern for better understanding and analyzing the service discovery model.

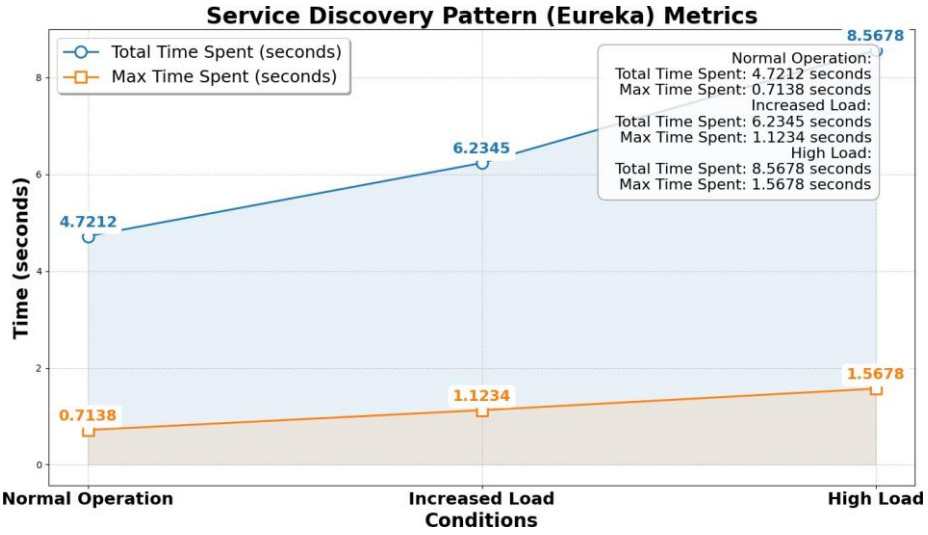

Figure 12: Visual representation of Service Discovery Pattern on test readings

This graph represents the common metrics as **Total Time Spent** (blue circle) and **Max Time Spent** (orange square) under normal operation flow of microservice architecture under Service Discovery pattern (Eureka). The Eureka server registers its clients, the microservices, with respective IP addresses for other services to discover and connect. With significant increase in requests under different conditions, the cumulative time taken to process all requests is **4.7212 seconds** under Normal Operations to further increase to **6.2345 seconds** under Increased Load, to further spike **8.5678 seconds** under High Load. Similarly, the maximum time to process a single request takes (max time spent) about **0.7138 seconds** under Normal Conditions to **1.1234 seconds** to **1.5678 seconds** respectively under increased load to high load.

6.3 Performance Comparison

As observed in the previous section, the test results explains that the response time is proportional to its size of payload. As we increase the condition of payload, the response time also increases with respect to the microservices. Let's analyze the latencies of both the models and conclude with which model is effective under different use cases. Here we compare both the readings of the model, considering the main metrics as **TOTAL TIME** of both circuit breaker and service discovery in figure below. The TOTAL TIME is the cumulative time taken to process the request and return back with a response. The blue line represents the time taken to respond under Circuit Breaker pattern while the green line represents the time to process request under the service discovery pattern.

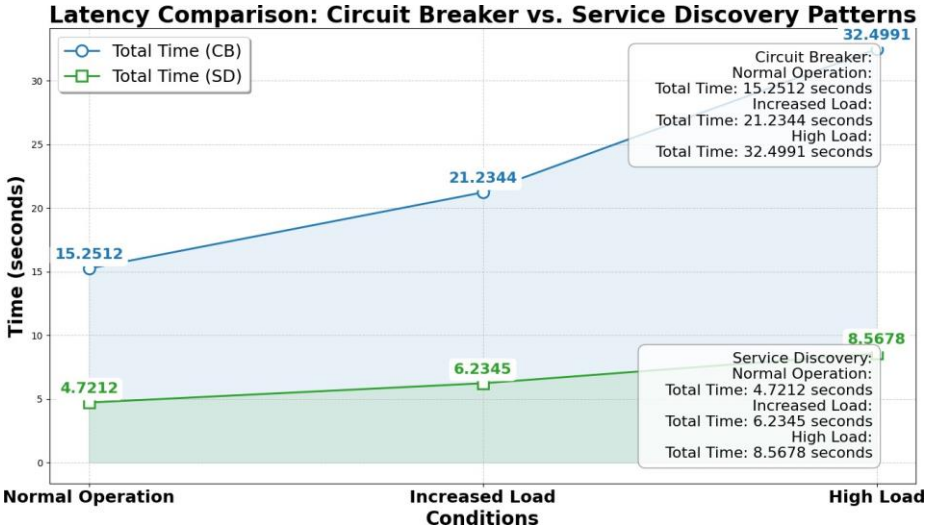

Figure 13: Analytical Comparison of Circuit Breaker and Service Discovery Patterns

6.4 Performance Combination

As we dive into the proposed approach of combining Service Discovery and Circuit Breaker patterns, we observe a significant improve in TOTAL TIME spent to complete the request under various load conditions compared to using each pattern individually. In individual pattern, as the graph depicts that under normal operation, there are increased load and high load conditions. The total time consumed to process the whole request in Circuit Breaker increases sharply from 15.2512 seconds to 32.4991 seconds, while in Service Discovery the time peaks from 4.7212 seconds to 8.5678 seconds, as shown in **Figure**.

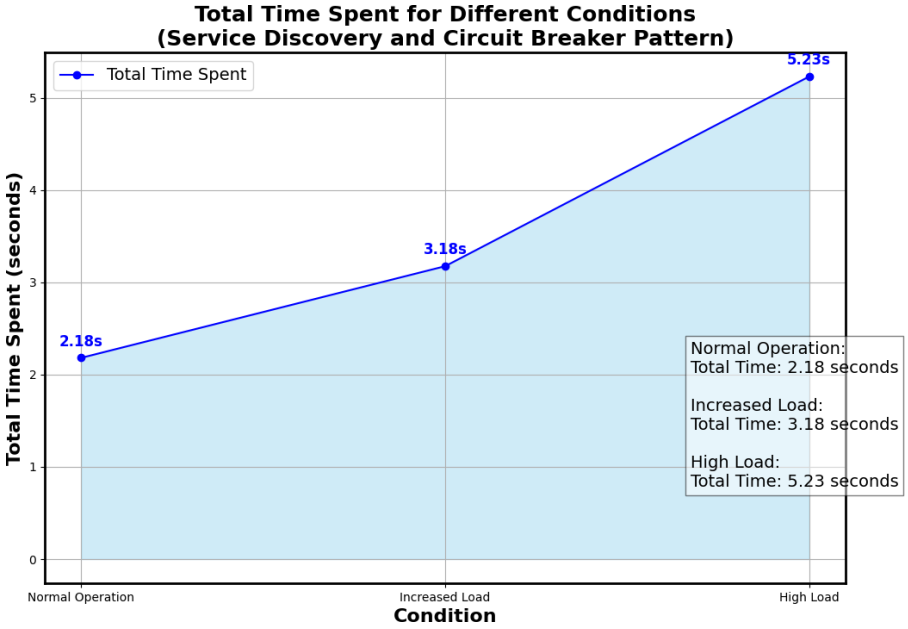

Figure 14: Visual representation of Service Discovery and Circuit Breaker Pattern on test readings

However, when we combine these two patterns into that architecture, we notice that under normal operation, the response time is 2.1821 which gradually increases to 5.2329 seconds under high load conditions. The collective working of both patterns ensures efficient resource utilization and maintenance of system's stability which results in reducing of latency and enhances the performance of system beyond what either pattern can achieve at individual level. Hence, by leveraging the strengths of both patterns, we achieve a more resilient and efficient system that is capable of now managing high amount of loads with much less latency and maintaining services effectively.

7 Conclusion and Future Work

This paper addresses the need for this research by posing the question, "Is it possible to reduce microservice response time latency by integrating and comparing the two widely used design patterns namely service discovery and circuit breaker?" While performing the experiments we discovered and proposed an architecture combining both patterns together. In the sections above we discussed the implementation of both service independently and also a combination working model on the backend services developed using Spring Boot framework. Comparing both the models we observed under normal operations to High load the response time increases sharply. Though both models show different response time respectively, we notice the service discovery pattern provides more promising results than circuit breaker**.**

But keeping in mind that both the patterns are useful under different use cases, we decided to combine both patterns together in order to achieve a more resilient system. Hence, the novelty of this research experimentation lies in combining both of these patterns together to get a better performance. By integrating both together, the observation exhibits high efficiency with in- creased performance. Now, we will take an average of the results of response time under High load condition of both patterns individually:

$$\text{Average High Load Response Time} = \frac{32.4991 + 8.5678}{2} \approx 20.5 \text{ seconds}$$

After combining both patterns, we come to an understanding that the response time observed under High Load is 5.23 seconds which can be said that the performance is enhanced by **4 times (4X)**. Hence, we can state the latency is reduced by one-fourth times the original. These improved results can ascribe the complementary functionality of both patterns which together optimizes the distribution of load and manages failures. But this result should be considered with caution. **In this research, we have not conducted an ex- tensive scalability analysis but the tools used in the experimentation are capable in taking care of the scalability feature of the microservices individually.** Building on the findings and results of this study, in future work we can integrate an adaptive machine learning algorithms with the hybrid model of service discovery and circuit breaker. With this proposal in picture, the application can be deployed across different operational environments working on real-time data and traffic. This ensures the autonomous optimizing of resource allocation with multiple response strategies based on patterns.

References

A, N. S. S. and Sebastian, S. (2023). Circuit breaker: A resilience mechanism for cloud native architecture, *2023 Global Conference on Information Technologies and Communications (GCITC)*, pp. 1–8.

Aibin, M. (2024). A guide to spring boot admin — baeldung.
URL: *https://www.baeldung.com/spring-boot-admin*

Awad, J. W. (2024). Microservices pattern: Service discovery — medium.
URL: *https://medium.com/@joudwawad/service-discovery-patterns-in-microservices-70b5d8ef0197*

Bhojwani, R. (2022). Microservice architecture and design patterns - dzone.
URL: *https://dzone.com/articles/design-patterns-for-microservices*

Bogner, J., Fritzsch, J., Wagner, S. and Zimmermann, A. (2019). Assuring the evolvability of microservices: Insights into industry practices and challenges, *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 546–556.

Daniel, J. a., Wang, X. and Guerra, E. (2024). How to design future-ready microservices? analyzing microservice patterns for adaptability, *Proceedings of the 28th European Conference on Pattern Languages of Programs*, EuroPLoP '23, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3628034.3628046*

Desina, G. C. (2023). Evaluating the impact of cloud-based microservices architecture on application performance.
URL: *https://arxiv.org/abs/2305.15438*

Gotin, M. (2018). About microservices, containers and their underestimated impact ...
URL: *https://arxiv.org/abs/1710.04049*

Gotin, M., Lösch, F., Heinrich, R. and Reussner, R. (2018). Investigating performance metrics for scaling microservices in cloudiot-environments, *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, Association for Computing Machinery, New York, NY, USA, p. 157–167.
URL: *https://doi.org/10.1145/3184407.3184430*

Khaleq, A. A. and Ra, I. (2021). Intelligent autoscaling of microservices in the cloud for real-time applications, *IEEE Access* **9**: 35464–35476.

Khan, S. (2024). Mastering spring boot service discovery and registration.
URL: *https://medium.com/@ksaquib/mastering-spring-boot-service-discovery-and-registration-using-spring-cloud-netflix-eureka-a-1eec70317b32*

Obregon, A. (2023). Spring boot admin — centralized management for your spring ...
URL: *https://medium.com/@AlexanderObregon/spring-boot-admin-centralized-management-for-your-spring-applications-70c25fba3bab*

Saboor, A., Mahmood, A. K., Hassan, M. F., Shah, S. N. M., Hassan, F. and Siddiqui, M. A. (2021). Design pattern based distribution of microservices in cloud computing environment, *2021 International Conference on Computer Information Sciences (ICCOINS)*, pp. 396–400.

Shadija, D., Rezai, M. and Hill, R. (2017). Microservices: Granularity vs. performance, *Companion Proceedings of The10th International Conference on Utility and Cloud Computing*, UCC '17 Companion, Association for Computing Machinery, New York, NY, USA, p. 215–220.
URL: *https://doi.org/10.1145/3147234.3148093*

Su, Y., Feng, D., Hua, Y. and Shi, Z. (2017). Predicting response latency percentiles for cloud object storage systems, *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 241–250.

Vale, G., Correia, F., Guerra, E., Rosa, T., Fritzsch, J. and Bogner, J. (2022). Designing microservice systems using patterns: An empirical study on quality trade-offs.

van Vessum, S. (2024). Amazon study: Every 100ms in added page load time cost 1
URL: *https://www.conductor.com/academy/page-speed-resources/faq/amazon-page-speed-study*

Wang, Y. (2019). Towards service discovery and autonomic version management in self-healing microservices architecture, *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, Association for Computing Machinery, New York, NY, USA, p. 63–66.
URL: *https://doi.org/10.1145/3344948.3344952*

Weilan, Pan, D. and Wang, P. (2022). Servicerank: Root cause identification of anomaly in large-scale microservice architectures, *IEEE Transactions on Dependable and Secure Computing* **19**(5): 3087–3100.

Černý, T. (2018). Aspect-oriented challenges in system integration with microservices, soa and iot, *Enterprise Information Systems* **13**: 1–23.