Navigating the Microservices Terrain
A Comprehensive Survey

CS60002: Distributed Systems

Group Number: 10

Group Members:

Pranav Mehrotra	20CS10085
Saransh Sharma	20CS30065
Pranav Nyati	20CS30037
Shreyas Jena	20CS30049
Anand Manojkumar Parikh	20CS10007

Contents

Chapter 1

Microservices : Introduction

With a steady increase in the computational requirements for performing critical, high-stake tasks and a proliferation of large-scale data centers capable of handling millions of requests per second, there has been a rise in the number of popular online cloud services that cater to diverse human activities. Many of these services are interactive and time-sensitive, implying the need for stringent limits on latency while consistently maintaining high performance and quality standards [7, 8]. To ameliorate the issues that come up with scaling cloud services both in terms of size and outreach, there has been a marked shift from intricate **monolithic services** that encapsulate entire application functionalities within a single binary to architectures consisting of numerous single-purpose **microservices** loosely coupled together. This transition is increasingly pervasive among major cloud providers like Google, Microsoft, Amazon, Twitter, Netflix, and Apple [9].

The rising popularity of microservices can be attributed to several factors. Firstly, they propose a **compositional software design approach**, streamlining the development processes by assigning each microservice responsibility for a specific subset of application functionalities. As cloud services become more feature-rich, the modular nature of microservices helps to manage system complexity efficiently. Moreover, microservices facilitate independent deployment, scaling, and updating of individual components. This leads to lesser drawn-out development cycles, reduces debugging complexity and enhances system flexibility to scale.

Fig 1.1 shows the deployment differences between a traditional monolithic service, and an application built with microservices. While the entire monolith is scaled out on multiple servers, microservices allow individual components of the end-to-end application to be individually scaled, with microservices of complementary resources allocated on the same physical server. Even though modularity in cloud services was already part of the Service-Oriented Architecture

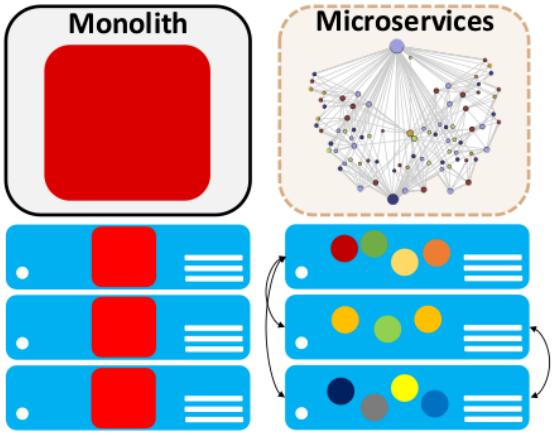

Figure 1.1: *Differences in the deployment of monoliths and microservices.*

(SOA) design approach [10], the fine granularity of microservices, and their independent deployment create hardware and software challenges different from those in traditional SOA workloads.

Secondly, microservices promote **programming language and framework diversity**, enabling each tier to be developed in the most suitable language, with only a common API required for inter-microservice communication, typically via remote procedure calls (RPC) [11] or a RESTful API. In contrast, monolithic architectures impose limitations on the choice of languages for development and complicate frequent updates, making them prone to errors. Lastly, microservices streamline correctness and performance debugging, as bugs can be isolated within specific tiers, unlike monolithic architectures where bug resolution often involves troubleshooting the entire service. This makes microservices particularly suitable for internet-of-things (IoT) applications that often host mission-critical computations, requiring correctness verification [12, 13].

Despite their advantages, the adoption of microservices represents a significant departure from traditional cloud service design methodologies due to their modular and distributed nature. This shift introduces new challenges, such as managing **communication and coordination** between multiple microservices, ensuring **data consistency and security** across distributed systems, **monitoring and troubleshooting** of individual services, and ensuring overall system **performance**, **scalability** and **availability**. To address these challenges, careful consideration must be given to the design choices made when implementing microservices architecture. The aim of this research survey is to provide a comprehensive understanding of the key choices made in microservices design and corresponding implications across diverse aspects, ranging

from communication control, data security and scalability to performance optimization and benchmarking.

We will direct our attention towards the following five research questions in order to thoroughly examine and review the existing literature.

- **RQ1:** What are the specific design challenges on microservice architecture, which the recent literature try to address?
- **RQ2:** Do the existing solutions on microservice architecture talk about a generic platform, or application-specific architecture? What are the goals for each of them?
- **RQ3:** What are the performance issues that are being addressed in the existing literature for microservices?
- **RQ4:** What are the core solution ideas being explored to address each of those issues that you could find out?
- **RQ5:** What are the limitations of the existing solutions that need further researches?

The existing solutions on microservice architecture can vary in terms of their focus. Some solutions discuss a generic platform for microservices, while others focus on application-specific architectures.

1. Generic Platform:
   Solutions that talk about a generic platform for microservices aim to provide a framework or infrastructure that can be used across different applications and industries. These platforms typically offer a set of tools, libraries, and services that help developers build, deploy, and manage microservices in a standardized and scalable manner. The goals of a generic platform include:
   - Promoting modularity and reusability: By providing a standardized framework, a generic platform enables developers to build modular microservices that can be easily reused across different applications.
   - Simplifying development and deployment: The platform offers tools and services that streamline the development and deployment processes, making it easier for developers to create and manage microservices.
   - Ensuring scalability and flexibility: A generic platform is designed to handle the scalability requirements of various applications, allowing them to scale up or down based on demand. It also provides flexibility in terms of technology choices and integration with other systems.

2. Application-Specific Architecture:
   On the other hand, some solutions focus on application-specific architectures for microservices. These solutions are tailored to address the specific requirements and challenges of a particular application or industry. The goals of an application-specific architecture include:

   - Optimizing performance and efficiency: By tailoring the architecture to the specific needs of the application, these solutions aim to achieve better performance and efficiency in terms of resource utilization, response time, and throughput.
   - Addressing domain-specific challenges: Different applications have unique requirements and challenges. An application-specific architecture takes into account these specific needs and provides solutions that are optimized for the particular domain.
   - Enhancing security and compliance: Certain industries, such as healthcare or finance, have strict security and compliance requirements. An application-specific architecture can incorporate specific security measures and compliance standards to ensure data protection and regulatory compliance.

Chapter 2

Design Challenges

Microservices architecture has gained significant traction in modern software development due to its ability to enhance scalability, flexibility, and agility. However, along with its advantages, microservices bring about unique design challenges that need to be carefully addressed. One of the key challenges lies in ensuring effective communication and coordination among the distributed services, as the decentralized nature of microservices can lead to increased complexity in managing interactions. Additionally, designing microservices to be independently deployable and scalable while maintaining consistency and reliability across the system poses another significant challenge. Moreover, managing data consistency and ensuring fault tolerance in a distributed environment further adds to the complexity of microservices design. In this rapidly evolving landscape, understanding and mitigating these design challenges are crucial for the successful adoption and implementation of microservices architecture in modern software systems.

In the upcoming sections, we will delve into various design challenges encountered in microservices architecture. We aim to examine the reasons behind their complexity and explore their implications on microservices systems.

2.1 Service Decomposition

One of the most important initial design challenges in microservice architecture design is the decomposition of a monolithic version of a service or application into smaller, independently deployable services. Such a decomposition is extremely important to enhance the scalability, flexibility, and maintainability of the system. With the rapid advent of cloud-based infrastructure innovations such as **Software-as-a-Service** and **Function-as-a-Service**, most of the tech-firms are transitioning from a monolithic design paradigm to a microservice-based paradigm. However, it has been reported by several leading tech companies (e.g., Amazon, Netflix, and Uber) [14] that service decomposition of legacy monolithic applications has been one of the

bottlenecks [15] in smoothly and rapidly transitioning to a microservice-based paradigm for the software industry. In microservice design, two main charateristics: **Cohesivity** and **Coupling** of services are primary aspects that have been widely used to analyze the extent to which a microservice is scalable, maintainable, and deployable. We discuss these aspects below.

Cohesivity of Services

Cohesivity refers to the degree to which elements within a service are related and work together to achieve a common goal. In the context of microservices, cohesive services exhibit a specific purpose, with minimal dependencies on external components. Several factors determine the extent of cohesion of services, such as Functional Cohesion (services should encapsulate related functionalities or features together), Temporal Cohesion (services should handle operations with similar temporal characteristics together, minimizing the need for frequent changes and updates to the service interface), etc.

Coupling of Services

Coupling refers to the degree of interdependence between services, where tightly coupled services have strong dependencies and are more challenging to modify or replace independently. Conversely, loosely coupled services have minimal dependencies and can evolve independently. Strong coupling is a characteristic of traditional monolithic applications, whereas current microservice design has evolved to adopt loose coupling of services due to extremely complex and continuously evolving design. Various forms of coupling exist in microservice architecture. For example, Service Coupling refers to the dependencies between services, such as direct service-to-service communication or shared data stores. Tight service coupling can hinder scalability and hinder deployment flexibility, due to increase in IPC. Similarly, Interface Coupling refers to the dependencies between service interfaces, where changes in one service's interface may impact other services consuming it.

Based on the above two factors, various automated and semi-automated approaches [16] and guidelines have been proposed to provide a systematic and hierarchical decomposition of monolithic applications.

These can broadly be grouped into the following approaches:

- **Domain-Driven Design (DDD):** Domain-Driven Design [17, 18] emphasizes modeling software systems based on the business domain. Services are decomposed around domain boundaries, known as bounded contexts, ensuring that each service encapsulates a specific

aspect of the domain. This approach ensures better alignment between the software architecture and the underlying business requirements, leading to more cohesive and loosely coupled services. For example, in an e-commerce application, services related to inventory management, order processing, and customer accounts may constitute separate bounded contexts. DDD promotes high cohesion and low coupling between services by aligning service boundaries with domain boundaries.

- **Functional Decomposition:** Functional decomposition [19] involves breaking down the application into smaller services based on functional components or features. Each service is responsible for implementing a specific function or feature of the application. For instance, in a social media platform, services for user authentication, content management, and notifications may be decomposed based on their respective functionalities. Functional decomposition enables teams to work independently on different features, fostering faster development cycles and easier maintenance.

- **Dataflow-driven Decomposition:** Dataflow-driven decomposition [20] focuses on partitioning the application based on data ownership and access patterns. Services are decomposed according to the data they manage, with each service having autonomy over its data store. For example, in a banking application, services for account management, transaction processing, and reporting may be decomposed based on their data requirements. Data-driven decomposition minimizes dependencies between services and allows for better scalability and performance optimization [21]. However, data consistency and synchronization challenges across distributed services must be handled with great care in such decomposition.
- **API-first Approach:** The API-first approach [22] is about designing clear and well-defined APIs before implementing the underlying services. Services are decomposed based on these APIs, ensuring that each service exposes a set of interfaces for interaction with other services or clients. For example, in a microservices-based e-commerce platform, product catalog, shopping cart, and payment processing services may expose RESTful communication APIs.

Thus, effective service decomposition is essential for designing scalable, maintainable, and resilient microservices architectures. Each method of service decomposition offers unique benefits and challenges, with implications on the cohesivity and coupling of services within the architecture. As an example, Domain-Driven Design (DDD) promotes high cohesion by aligning service boundaries with domain boundaries, while the API-first approach encourages loose coupling through well-defined interfaces. Understanding the trade-offs associated with different decomposition methods is crucial for architects and developers to make informed decisions in

designing microservices architectures that meet the requirements of the business domain.

2.1.1 Blueprint [1]

In the paper [1], the authors argue that a key requirement for microservice experimentation is the ability to rapidly reconfigure applications and to iteratively Configure, Build, and Deploy (CBD) new variants of an application that alter or improve its design. Towards this goal, the central goal of researchers is to easily explore the design space of microservices, allowing them to move between different implementations and deployment configurations of the same application quickly and easily.

However, enabling design space exploration is difficult for several reasons. First, the design space of microservice systems is enormous. Application design, configuration, and deployment choices vary along several dimensions: specific patterns in the application logic and the inclusion of particular features. Second, existing microservice systems are implemented as point solutions in this vast design space and are not designed to be reconfigurable or extensible. Thus, the entire burden for moving from one implementation to another falls on the researcher as they have to make deep modifications to microservice applications to fulfill their use-case.

The proposed solution Blueprint is a microservice development toolchain designed for rapidly Configuring, Building, and Deploying (CBD) microservices. Blueprint enables its users to easily mutate the design of an application and generate a fully-functional variant that incorporates their desired changes. The key insight of Blueprint is that the design of a microservice application can be decoupled into : (i) the application- level workflow, (ii) the underlying scaffolding components such as replication and auto-scaling frameworks, communication libraries, and storage backends, and (iii) the concrete instantiations of those components and their configuration.

The key contributions of Blueprint are as follows :

(i) To mutate off-the-shelf microservice applications with just a few LoC, to swap an instantiation, enable or disable scaffolding (e.g. replication), or change backends (e.g. database).

(ii) To develop new application workflows and generate runnable systems. Instead of binding workflow code to specific scaffolding and instantiations (e.g. the choice of RPC framework), those are declared separately with 10s of LoC, and incorporated by Blueprint at compile time.

(iii) introduce support for new instantiations (e.g. an experimental RPC framework) or scaffolding concepts and transparently apply them to existing applications.

2.2 Security Concerns

Microservice architecture, while offering numerous advantages in terms of scalability and agility, introduces significant security challenges. One of the primary concerns stems from the sheer volume of services inherent in this architecture, leading to an expanded attack surface. With each service functioning as an independent entity, there's an increased risk of vulnerabilities across various components, potentially exposing the entire system to exploitation. Moreover, the decentralization of functionality means that traditional security measures like centralized access control become more complex to implement effectively.

With each service exposing APIs for communication, there's a higher likelihood of vulnerabilities being exploited, leading to potential breaches. Additionally, the lack of a centralized security mechanism across services can make it challenging to enforce consistent security policies, leaving gaps that attackers could exploit.

Another major security concern with microservices revolves around the issue of establishing trust between services. With numerous services communicating over networks, ensuring secure communication channels becomes paramount. Services often need to communicate with each other to fulfill user requests. However, establishing and maintaining trust between these services can be complex, especially in scenarios where services are developed and maintained by different teams or even organizations. Without proper authentication, authorization, and secure communication channels, malicious actors could impersonate legitimate services or eavesdrop on sensitive data exchanged between services [23]. However, managing authentication and authorization across these distributed components can pose significant challenges. Additionally, the lack of a shared database among microservices complicates data security and access control mechanisms, requiring robust solutions for protecting sensitive information [24].

Furthermore, microservice architectures introduce complexities in secret management. With each service potentially requiring access to various credentials and keys, securely storing and managing these secrets becomes crucial. However, traditional approaches to secret management may not scale well in a microservices environment, necessitating novel solutions to ensure the confidentiality and integrity of sensitive information.

Moreover, the dynamic nature of microservices, where instances can be spun up or down dynamically to handle varying workloads, poses challenges for security monitoring and management. Traditional security tools and practices designed for monolithic architectures may struggle to keep up with the rapid pace of change in a microservices environment. This dynamicity also

extends to container orchestration platforms like Kubernetes, where managing security configurations across a large number of containers becomes a non-trivial task, potentially leading to misconfigurations and vulnerabilities [23].

Lastly, the complexity introduced by polyglot architectures, where services may be implemented using different programming languages and frameworks, adds another layer of security concern. Each technology stack comes with its own set of security considerations, making it difficult to ensure consistent security posture across the entire architecture [23].

Addressing these security concerns requires a holistic approach that encompasses secure design principles, robust authentication and authorization mechanisms, real-time monitoring and response capabilities, and ongoing security testing and validation. Collaborative efforts between industry practitioners and academic researchers are essential to develop effective security solutions that can adapt to the evolving landscape of microservices.

Another important aspect of security comes with the growing popularity of Cloud Native Applications. Cloud Native Applications (CNA) represent a modern approach to software development, leveraging microservices and cloud design patterns to deliver increased productivity, scalability, and cost-effectiveness. At the core of CNA architecture are microservices, which enable the decomposition of monolithic applications into smaller, independent components, each with its specific functionality. While CNAs offer numerous benefits, they also introduce several security concerns, particularly about the implementation and management of microservices.

One major security concern associated with microservices in CNA deployments is the distributed nature of communication. Microservices interact with each other over networks, often using lightweight communication protocols such as REST and Thrift. This distributed communication model exposes microservices to various network-based attacks, including Man-in-the-Middle (MiTM) attacks and session hijacking. Moreover, the decentralized nature of communication pathways increases the attack surface, making it challenging to monitor and secure interactions between microservices effectively.

Furthermore, the trust relationships among inter-communicating microservices within a common security trust domain pose potential security risks. While microservices are designed to operate within a trusted environment, a compromised microservice could exploit its privileged access to propagate attacks across the application ecosystem. Implementing robust access controls and authentication mechanisms becomes crucial to prevent unauthorized access and limit the impact of security breaches.

Lastly, implementing comprehensive logging, monitoring, and auditing mechanisms can enhance visibility into the security posture of microservices. By collecting and analyzing logs and metrics from microservices, abnormal behavior indicative of security threats can be detected and responded to promptly. Additionally, integrating security information and event management (SIEM) systems can centralize security monitoring and enable proactive threat detection and

response.

In subsequent sections, we will delve into each of these strategies in detail, discussing their implementation, benefits, and best practices for securing microservices within Cloud Native Applications.

2.2.1 Approaching towards more secure Microservice environment

After delving into the core importance and complexities surrounding security in microservices, we must examine existing solutions or methods that strive to enhance the security of microservices.

[25] proposes an approach to make microservices more resilient against application-layer denial-of-service (DoS) attacks. The key objective is to develop an unsupervised, non-intrusive, and application-agnostic detection mechanism that can identify when a microservice is under attack by monitoring resource utilization patterns.

The proposed methodology leverages the distributed nature of microservices to isolate and defend against attacks. It continuously monitors the CPU and memory utilization of each microservice pod (container) using a first-order autoregressive model. By comparing the observed utilization to historical profiles of normal behavior, the system can detect when a pod is exhibiting anomalous resource usage indicative of an attack.

Upon detecting an attack, the system uses a "service fissioning" approach to quarantine the affected pod on a dedicated node. It then progressively splits the pod's users across new quarantined replicas, ultimately isolating the attacker. This allows the system to minimize the impact on legitimate users while identifying the source of the attack.

The paper presents a prototype implementation (Figure: 2.1) of this approach built on top of the Kubernetes container orchestration platform. Experimental results show that the proposed defense mechanism can reduce the latency experienced by legitimate users by up to 3x compared to having no protection in place during an attack. This demonstrates the effectiveness of the unsupervised detection and targeted quarantine/isolation strategy in making microservices more resilient to application-layer DoS attacks.

The paper titled: "Enhancing Microservices Security with Token-Based Access Control Method" [26] tries to solve another important aspect of microservices' security by identifying that the decentralized nature of microservices architecture increases the number of potential entry points for attackers, making system security more difficult. A vulnerability in a single microservice can have cascading effects on the entire system, so ensuring the security of all services is crucial.

To address this issue, the paper proposes a token-driven access control method for microservices. The key aspects of their approach are:

External authentication and internal authorization: The method separates the authenti-

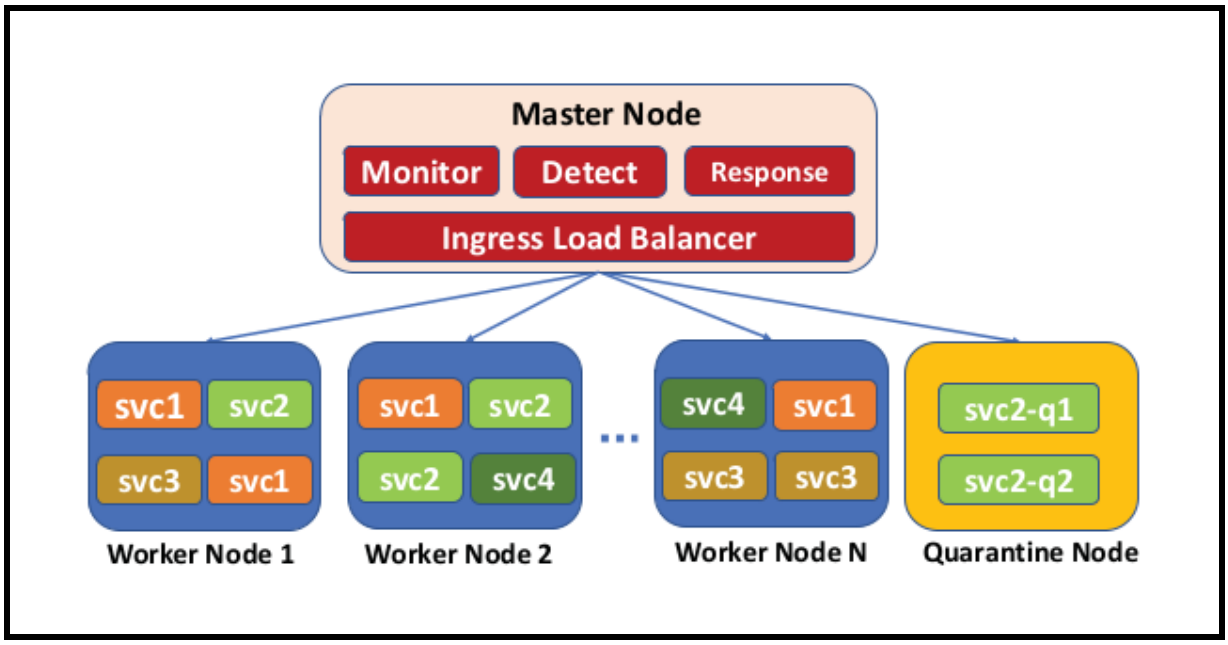

Figure 2.1: *Prototype Implementation*

cation and authorization processes. External authentication is handled by an authentication service that issues access tokens to clients. Internal authorization is handled by separate authorization services (Figure: 2.2) that validate the access tokens and authorize requests to the resource microservices. Decentralized access control: By distributing the access control responsibility across multiple microservices, the method enhances the security of the decentralized microservices architecture. This reduces the risk of a single point of failure or attack and minimizes the impact of a breach or compromise of one microservice on the rest of the system. Through experimental evaluation, the authors show that their proposed method is more efficient under low and medium loads, while providing enhanced security under high loads compared to centralized access control approaches. The decentralized nature of their solution helps prevent unauthorized access and reduces the risk of attacks on the microservices.

2.3 Network Challenges

Microservices architecture has emerged as a popular paradigm for building scalable and resilient software systems composed of loosely coupled, independently deployable services. Central to the effectiveness of microservices is the efficient communication between these distributed components. Communication in microservices encompasses various aspects, including synchronous and asynchronous patterns, inter-service protocols, service discovery mechanisms, load balancing strategies, and fault tolerance mechanisms. Synchronous communication, often facilitated

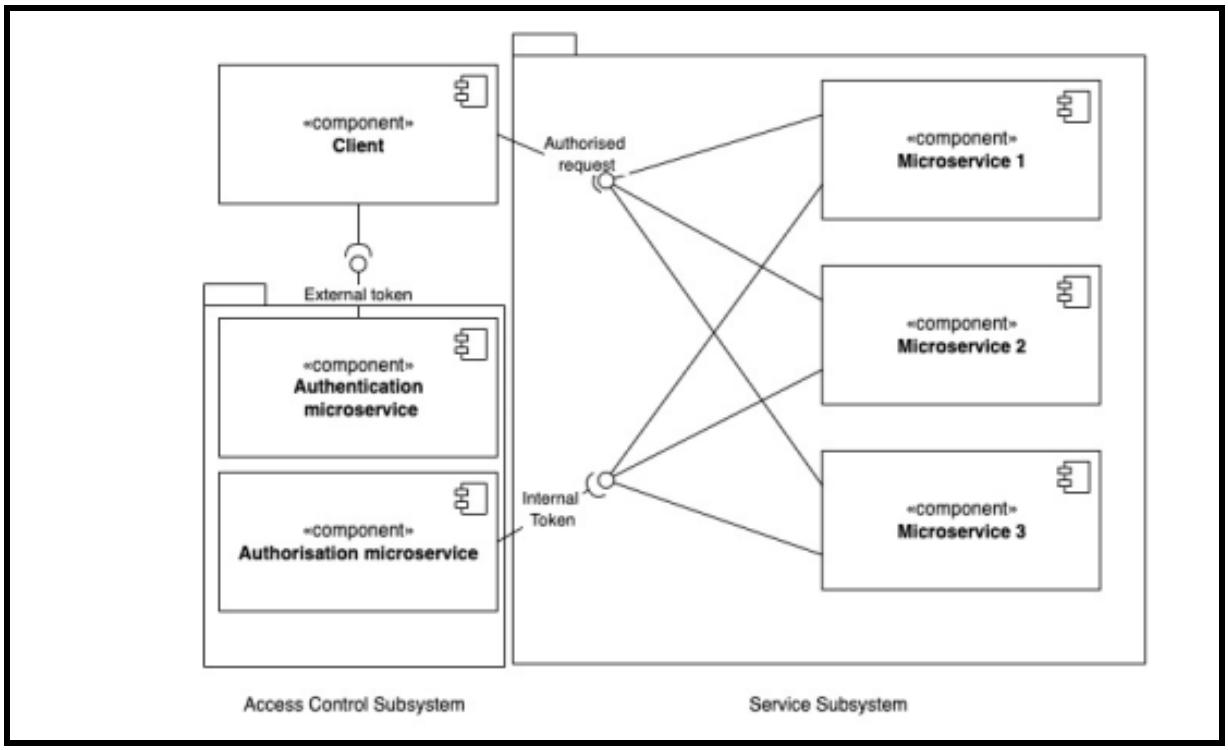

Figure 2.2: *Diagram of the components for the proposed decentralized access control in microservices*

by RESTful APIs or gRPC, allows services to interact in a request-response manner, while asynchronous communication (Figure: 2.3), enabled by message brokers like RabbitMQ or Apache Kafka, supports decoupled interactions and event-driven architectures. Service discovery mechanisms such as DNS-based discovery or service registries play a vital role in enabling dynamic service location and addressing, ensuring seamless communication within the microservices ecosystem.

Synchronous communication, such as RESTful APIs, typically involves a request-response interaction model, where a client sends a request to a service and waits for a response before proceeding further. REST, or Representational State Transfer, is an architectural style for designing networked applications. RESTful APIs (Application Programming Interfaces) are built on top of the HTTP protocol and use standard HTTP methods such as GET, POST, PUT, DELETE to perform CRUD (Create, Read, Update, Delete) operations on resources. REST APIs are stateless, meaning each request from a client contains all the necessary information for the server to fulfill it, and responses typically include data in formats like JSON or XML. RESTful principles emphasize simplicity, scalability, and compatibility with existing web standards, making them suitable for building web services that are easy to understand, maintain, and integrate with other systems.

One of the primary challenges with synchronous communication is its blocking nature.

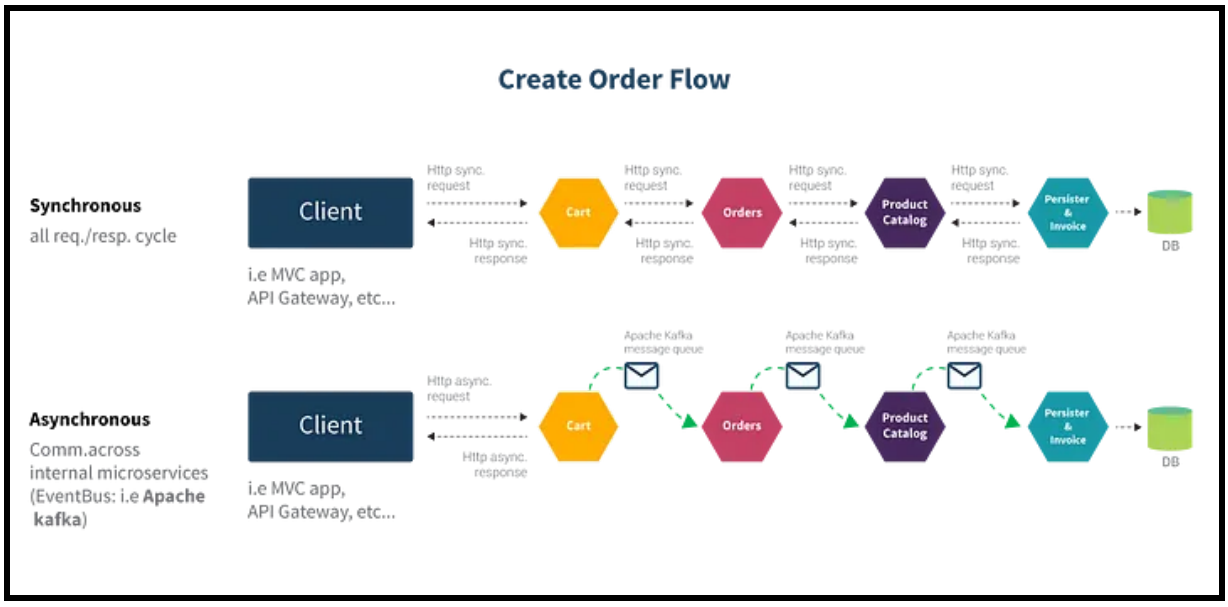

Figure 2.3: *Synchronous vs Asynchronous communication*

In a synchronous system, if a service is slow to respond or becomes unavailable, it can lead to increased latency and potential bottlenecks, affecting the overall performance and responsiveness of the system. Additionally, synchronous communication can be more susceptible to cascading failures, where a failure in one service can propagate to other services due to blocking calls, resulting in degraded system reliability.

On the other hand, asynchronous communication, facilitated by message brokers like RabbitMQ or Apache Kafka, allows services to exchange messages without waiting for an immediate response. While asynchronous communication offers benefits such as decoupling and improved scalability, it also introduces its own set of challenges. One challenge is message durability and consistency. Since messages are typically stored in queues or topics before being processed, ensuring message durability and consistency becomes essential, especially in scenarios where message processing can fail or services need to maintain consistency across distributed transactions.

gRPC, a common framework highly used by microservices, is a high-performance Remote Procedure Call (RPC) framework developed by Google. gRPC uses HTTP/2 as its transport protocol and Protocol Buffers (protobuf) as its interface definition language (IDL) for describing service interfaces and message payloads. Unlike REST, which is based on textual formats like JSON or XML, gRPC messages are binary-encoded, resulting in smaller message sizes and improved efficiency in data transmission. gRPC is particularly well-suited for building microservices that require high performance, low latency, and real-time communication, such as streaming applications, microservices communicating with IoT devices, or microservices within performance-sensitive environments.

For gRPC, a significant challenge lies in its complexity, particularly in comparison to REST. The protocol buffer-based serialization used by gRPC can be more intricate for developers accustomed to working with JSON, which is the common format utilized in REST APIs. This complexity may result in a steeper learning curve and potentially longer development cycles. Additionally, while gRPC's reliance on HTTP/2 for communication offers advantages in terms of performance and efficiency, it can present challenges in environments where legacy systems or network intermediaries do not fully support the protocol [27].

Both gRPC and REST API architectures come with their own set of challenges that need to be carefully addressed. On the other hand, REST APIs face challenges related to scalability, versioning, and data transfer efficiency. Managing a large number of endpoints in complex systems can become cumbersome and may lead to issues with scalability and maintainability. Versioning REST APIs also poses challenges, as introducing changes or new features while maintaining backward compatibility can result in version proliferation or breaking changes if not managed carefully. Furthermore, REST APIs may suffer from over-fetching or under-fetching of data, where clients receive more or less data than necessary, impacting both performance and bandwidth usage [28].

With growing interest in RPCs i.e. remote procedural calls for inter-service communication in modern architectures, it becomes imperative for us to analyze the challenges associated with RPC framework. "Dagger: Towards Efficient RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs"[29] analyzes that even though RPC request and response payloads are relatively small in typical data center applications, ranging from hundreds of bytes to a few kilobytes, in microservices they are even smaller, with more than 70 percent of RPC requests being smaller than 256B and almost all requests within 1280B. These tiny messages introduce high pressure on networking stacks at all levels, and commodity networking systems cannot efficiently handle this traffic due to high per-packet overheads. Optimizing RPCs in microservices is crucial for meeting the stringent QoS requirements and achieving low latency. Traditional RPC frameworks were not designed with the unique characteristics of microservices in mind, such as fine-grained and concurrent workloads, diverse performance requirements, and the need for runtime reconfigurability.

The paper "The NEBULA RPC-Optimized Architecture" [30] highlights the need for RPC optimization in the context of microsecond-scale RPCs, which are prevalent in performance-critical tiers like in-memory data stores and microservices. These RPCs have runtimes of only a few microseconds, making them highly susceptible to even small latency overheads that were previously negligible.

Moreover, the high communication-to-computation ratio in microservices architectures means that networking becomes the key performance determinant. As the authors of the NEBULA paper[30] point out, the growing network bandwidth in modern data centers can lead to

memory bandwidth interference, where incoming network traffic competes with the application's memory requests, causing queuing effects that degrade the tail latency of these microsecond-scale RPCs, thereby affecting the performance of microservice architecture.

In subsequent sections, we will explore various endeavors aimed at enhancing RPC communication efficiency, minimizing RPC calls within a microservices architecture, introducing alternative frameworks for network issue monitoring, and optimizing network latencies.

2.4 Load Balancing and Resource Allocation

Over the past decade, leading online companies such as Uber, Netflix, Amazon, Twitter, PayPal, eBay, and many more have migrated from monolithic to microservice architecture. These frameworks are extremely user-intensive, receiving tens of thousands, if not millions of client requests per minute. At its core, load balancing is the task of distributing network traffic equally across a pool of resources to ensure that no single instance is overwhelmed with too much traffic and it also aids in improving the availability, scalability, security, and performance of such a service. The load balancer is the first point of contact between users and the application, which directly affects the latency and response time for the user and also the computational and networking loads of the underlying servers, making it a critical challenge to devise neat, efficient, scalable and secure load balancing systems for distributed microservice architectures.

First, we discuss the most fundamental piece of load balancers: the load balancing algorithm, which essentially maps client requests to servers. For higher availability and fault tolerance, resources are generally replicated across servers (in both monolithic and microservice architectures). The load balancer must efficiently and evenly distribute requests to access a distributed resource. This task becomes even more interesting for microservices, since each module of the service manages a different subsystem, possibly holding a different resource, while itself being replicated across servers. Some general techniques for load balancing are Round Robin, Weighted Round Robin (more requests are directed to servers with higher capacity), hashing methods (e.g. consistent hashing), least connection methods, least response time methods, etc.

However, such simplistic methods do not always yield the best results for microservices because of many reasons. For e.g., the traffic is unpredictable, and sudden spikes in user activity must not stall/overload the system. Also, many microservice instances are short-lived and might be reconfigured multiple times a day, compared to configuration updates typically only done a few times a month in monolithic services. The load balancer must be automated to adapt to these changes dynamically without any manual intervention from system administrators. [31] presents a customized definition of server load, taking into account CPU, disk, and memory utilization rates and the number of server connections which represents the load more accurately in real-time and shows that their custom algorithm shares load more evenly for a microservice

cluster, compared to classical techniques. [32] leverages DHTs (Distributed Hash Tables) and improve upon existing consistent hashing techniques to come up with the first P2P (peer-to-peer) algorithm for simultaneously achieving logarithmic degree (no. of neighbors each node must track), logarithmic lookup cost (no. of hops required for fetching any data), and constant time load balance (determining the target server for a request). [33] devises a novel randomized algorithm to choose n points on a circle with unit circumference where each new server (or point) can learn its position by knowing the positions of a minimum number of other servers (or points) while minimizing the variance of the lengths of arcs on the circle made by these points. Client requests can now be randomly mapped on this circle, and the next clockwise/counter-clockwise server is chosen. The two techniques above provide detailed mathematical proofs and algorithmic analysis to support their claims, and even though they never mention microservices explicitly, they are indeed of interest for balancing load within the microservice for communication amongst different modules (east-west traffic).

The strongest motivation for distributed computing is decentralization, increasing scalability, availability, and throughput. However, choosing decentralized load-balancing methods for north-south traffic (from client requests to microservice modules) comes with even more challenges. Centralized load balancing schemes provide better scheduling policies, and resource management, and generally have better algorithmic and message complexity since a single (or leader) load balancer drives all operations, but this partially defeats the purpose of distributed computing. Their decentralized counterparts are more reliable and scalable, but also more inefficient. [4] proposes HEELS, a scheme combining the best of both centralized and decentralized techniques. HEELS is implemented using eBPF as a Layer 4 load balancer which is compatible with modern public cloud systems and can be deployed with no kernel modifications and minimal performance overhead.

Load balancing in microservices comes with some challenges not immediately found in monolithic services, which must be managed separately. One such challenge is the chaining of load. Microservice modules each perform logically separate tasks and as expected, are sometimes non-trivially dependent on each other. Variation of load on a particular module can directly affect others dependent on it, and so forth recursively, causing a situation called load chaining. Failure of a particular subsystem (through no fault of its own, but simply because it was overloaded with requests) can render many other subsystems useless. It is the responsibility of the load balancer to avoid such undesirable situations by careful examination of dependencies before making balancing decisions. [34] utilizes a dependency graph for describing interdependencies amongst a microservice's modules. They provide a provably polynomial time approximation algorithm to solve the QoS-aware load balancing problem (load balancing to maximize the Quality of Service), which is known to be NP-hard and shows promising results compared to current heuristic approaches. [35] presents TCLBM, a task chain-based load balancing algorithm for

microservices, that computes the resource usage for each system instance and the number of data transmissions between machines to analyze chaining of load to avoid load imbalance amongst candidate instances.

Apart from these core functionalities, many others are built into load balancers of modern microservice architectures. Being at the front of the service interface and exposed to users, both monolithic and microservice architectures implement security checks such as firewalls as part of their load balancers, thereby preventing attacks and breaches at the source itself, without allowing malicious access to deeper levels of the architecture, which might potentially cause more severe harm like data theft from the service. For microservice architectures, dynamic system updates are very frequent, meaning the load balancers must adapt to modules being launched and/or killed frequently. Also, having more services means more probability of some of them failing due to faults. [36] describes a self-healing microservice framework that autodetects faults and updates through service discovery without manual intervention.

2.5 Fault Tolerance

Fault tolerance is the ability of a system to continue operating properly in the event of the failure of some of its components. This is an important consideration in microservices, as the distributed nature of microservices can introduce new points of failure compared to a monolithic architecture.

Fault tolerance is a key challenge when it comes to microservices architectures for several reasons:

1. Distributed Nature of Microservices:
   Microservices are designed to be independently deployable and scalable components, often running in separate processes or containers. This distributed nature introduces more potential points of failure compared to a monolithic application, where failures may be more localized.

2. Increased Complexity:
   Coordinating and managing the interactions between multiple microservices adds complexity to the overall system. Failures in one microservice can potentially cascade and affect other dependent services, making it more difficult to maintain reliability.

3. Network Failures:
   Microservices communicate over the network, which introduces the risk of network-related failures, such as connection timeouts, network partitions, or high latency. These network-level issues can lead to service disruptions or inconsistent behavior.

4. State Management:
   Microservices often maintain their own state, which can make it harder to ensure consistency and durability of data in the face of failures. Coordinating state across multiple services requires careful design and additional mechanisms like distributed transactions or event-driven architectures.

5. Increased Operational Complexity:
   Managing, scaling, and monitoring a distributed system of microservices is more operationally complex compared to a monolithic application. Automating deployment, scaling, and failure recovery processes becomes crucial for maintaining fault tolerance.

6. Debugging and Troubleshooting:
   When a failure occurs in a microservices-based system, it can be more difficult to diagnose the root cause due to the distributed nature of the system and the potential for cascading failures.

   To address these challenges, microservices architectures often require additional patterns and techniques, such as service discovery, circuit breakers, retries, fallbacks, and sophisticated monitoring and logging systems. Careful design and implementation of these fault tolerance mechanisms are essential for building reliable and resilient microservices-based applications.

Dynamic Microservices to Create Scalable and Fault Tolerance Architecture [37] proposes a dynamic microservices architecture that aims to improve scalability and fault tolerance. Key aspects related to fault tolerance include:

1. Dynamic distribution of tasks between clients: The server can dynamically assign tasks to available clients, and if a client fails or becomes unresponsive, the server can take over and execute the task itself. This improves fault tolerance by providing redundancy and the ability to gracefully handle client failures.

2. Orchestration for extended microservices: When a microservice needs to call another microservice to complete a task, the server handles the orchestration. This avoids direct client-to-client communication, which could be vulnerable to failures. The server can find an available client to execute the dependent task and redirect the response back to the original client.

3. Monitoring and handling of client failures: The server continually monitors the status of connected clients. If a client becomes unresponsive, the server will attempt to re-verify its availability, and if it remains unresponsive, the server will remove it from the list of available clients and take over its tasks.

4. Encryption and secure communication: The paper mentions using encryption and key exchange to secure the communication between the clients and the server, which helps protect the system against attacks that could compromise fault tolerance.

   By implementing these fault tolerance mechanisms, the proposed microservices architecture aims to create a more scalable and resilient system that can gracefully handle failures of individual components (clients) without compromising the overall functionality of the application.

Meanwhile, Detection of Faults in Microservices using Petri Nets [**?**] proposes a novel approach to detect faults in microservices during the early requirements and design phases, rather than waiting until runtime or deployment. The key elements of their methodology are:

1. Mapping the microservice requirements to Netflix's Conductor specification, which is a JSON-based language for defining microservice workflows.

2. Converting the Conductor specification into a timed Petri net model using the Conductor2PN tool.

3. Performing traversal analysis on the Petri net model to derive various properties expressed as Computation Tree Logic (CTL) formulas. These properties include deadlock detection, liveness checking, and verification of bounded response times.

4. For simple Petri net models (less than 1000 states), the CTL properties can be directly model-checked. For more complex models, the properties are encoded as constraints and checked using an SMT solver like Z3.

5. By analyzing the counterexamples generated when the properties are violated, the approach is able to identify various types of faults in the microservice design, such as incorrect sequence of asynchronous operations, missing information, long response times, timeouts, and service failures.

   The key advantage of this approach (Figure: 2.4) is that it allows detecting and correcting microservice faults early in the development lifecycle before they manifest in the deployed system. This can lead to significant savings in terms of development effort and resources.

2.6 Benchmarking

We explore the following setups specifically created for benchmarking microservices :

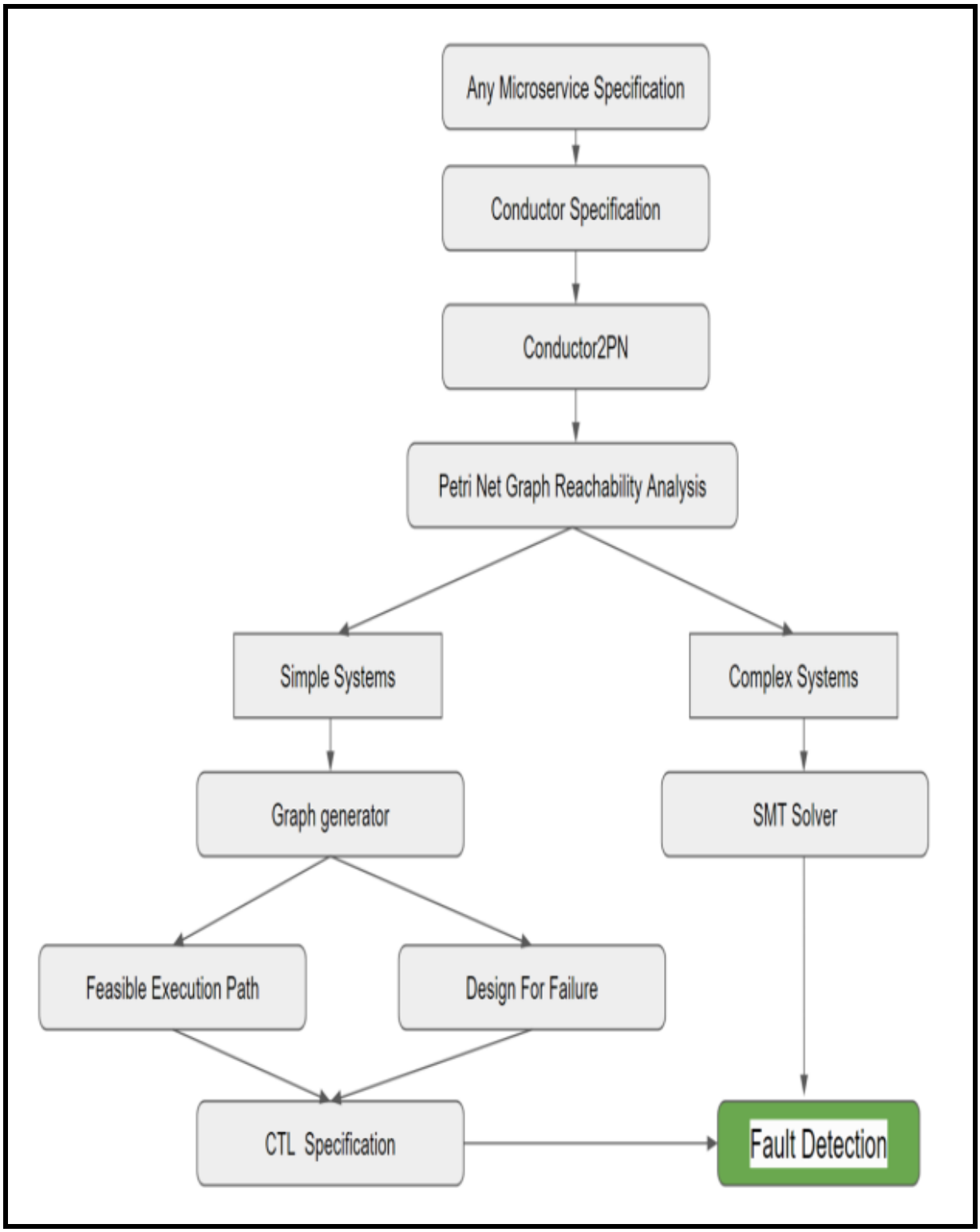

Figure 2.4: *Proposed Architecture*

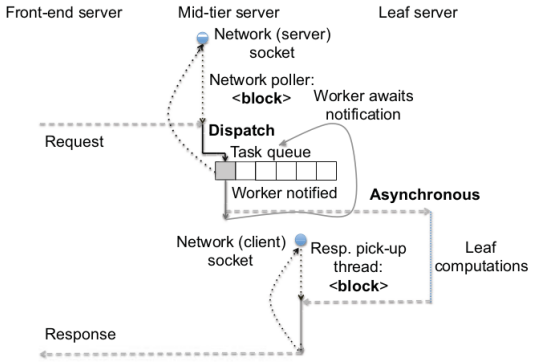

Fig. 8: μSuite's mid-tier microservice design.

Figure 2.5: *μSuite's mid-tier microservice design.*

2.6.1 μSuite [2]

Modern applications have undergone a design shift from monolithic systems to instead comprise numerous, distributed microservices interacting via Remote Procedure Calls (RPCs). These microservices require **single-digit millisecond RPC latency goals** — much tighter than prior monolithic systems that must meet greater than **100 ms** latency targets [38]. Sub-ms-scale OS/network overheads that were once insignificant for such monoliths can no longer be ignored in the sub μsecond limits required by individual microservices. It is therefore essential to characterize the influence of OS- and network-based effects on microservices.

However, academic benchmark suites such as CloudSuite [38] are unsuitable for this purpose as they : (1) use monolithic rather than microservice architectures, and (2) largely have request service times greater than 100 ms. Moreover, the hierarchical structure of microservices implies that individual containers at different levels face varying traffic levels. The authors claim that contrary to prior works focusing on leaf-level servers [39], there is a need to analyse the **mid-tier servers**, which receive a bulk of the incoming and outgoing requests.

The paper [2] proposes μSuite, a benchmark suite which comprises four services composed of microservices: **image similarity search**, **protocol routing** for key-value stores, **set algebra** on posting lists for document search, and **recommender systems**. They characterize each service in terms of its constituent mid-tier and leaf microservices. All of the μSuite benchmarks are built using gRPC. The key components of the design used for building each of μSuite's microservices (as illustrated in Fig. 2.5) are :

(i) Leveraging a **thread-pool** architecture that concurrently executes requests by judiciously "parking" and "unparking" threads to avoid thread creation, deletion, and management

overheads.

(ii) Blocking on the front-end network socket by using **network poller** threads awaiting new work from the front-end via blocking system calls, yielding CPU if no work is available.

(iii) Setting up **asynchronous communication with leaf microservers**, which allows the mid-tier microservers to process successive requests after sending requests to leaf microservers.

(iv) **Dispatch-based processing of front-end requests**, which separates responsibilities between network threads, accepting new requests from the underlying RPC interface, and worker threads, which execute RPC handlers.

2.6.2 DeathStarBench [3]

While μSuite quantifies the system call, context switch, and other OS overheads in microservices, the paper [3] proposes a new benchmark called DeathStarBench which focuses on large-scale applications with tens of unique microservices, allowing us to study effects that only emerge at large scale, such as network contention and cascading QoS violations due to dependencies between tiers, as well as by including diverse applications that span social networks, media and e-commerce services, and applications running on swarms of edge devices.

DeathStarBench includes a social network, a media service, an e-commerce site, a banking system, and IoT applications for coordination control of UAV swarms. It follows the given design principles :

(i) **Representativeness** : Code is built using popular open-source applications deployed by cloud providers, such as NGINX and MongoDB.

(ii) **End-to-end operation** : It implements the full functionality of a service from the moment a request is generated at the client until it reaches the service's backend or returns to the client.

(iii) **Heterogeneity** : The suite uses applications in low and high-level languages including C/C++, Java and Javascript.

(iv) **Modularity** : For designing, they follow the principle that the software architecture of a service follows the architecture of the company that built it in the design of the end-to-end

applications.

(v) **Reconfigurability** : Their intercommunication API allows swapping out microservices for alternate versions, with small changes to existing components.

The key contributions of this paper is that they explore the tail at scale effects of microservices in real deployments with hundreds of users, and highlight the increased pressure they put on performance predictability.

Chapter 3

Performance Dynamics in Microservices

Performance in microservices refers to the ability of a system to efficiently handle and process requests, deliver responses promptly, and maintain optimal levels of throughput and latency. It encompasses factors such as response time, resource utilization, scalability, and reliability. Performance is vital in microservices because it directly impacts user experience, system responsiveness, and overall efficiency. Efficient performance ensures that microservices can handle varying workloads, scale dynamically to meet demand and maintain consistent service levels even under heavy loads. Poor performance can lead to bottlenecks, delays, and service disruptions, ultimately affecting customer satisfaction and business operations. Therefore, optimizing performance is critical for maximizing the benefits of microservices architecture and achieving desired business outcomes.

This section will delve into various significant performance challenges that require scrutiny, as well as existing solutions and their limitations.

3.1 Communication Latency

In the realm of microservices architecture, communication between services often relies on Remote Procedure Calls (RPCs) as a core mechanism. RPCs facilitate seamless interaction between microservices, enabling them to communicate across distributed environments. Through RPCs, microservices can invoke functions or procedures hosted on remote servers, allowing for efficient exchange of data and execution of tasks.

Efficient communication is paramount for the optimal performance of microservices. By employing RPCs, microservices can leverage lightweight protocols like gRPC or JSON-RPC, which minimize overhead and latency. This streamlined communication ensures swift data

transmission and rapid response times, vital for maintaining the agility and scalability inherent in microservices architecture.

Moreover, efficient communication plays a pivotal role in enhancing the overall resilience and fault tolerance of microservices. By promptly propagating requests and responses, RPCs enable services to gracefully handle failures and adapt to fluctuating workloads. This robust communication mechanism fosters reliability and ensures seamless operation, even in the face of network disruptions or service outages.

3.1.1 RPC Optimization

The publication titled "**Dagger: Efficient and Fast RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs**" underscores the growing need for streamlined communication within distributed systems, particularly in cloud-based microservices environments. While Remote Procedure Calls (RPCs) serve as a common means of facilitating inter-service communication, conventional RPC frameworks often suffer from notable drawbacks such as latency issues and limited throughput, posing challenges to system performance and scalability.

The primary focus of the paper is to tackle the inefficiencies associated with RPCs in **cloud-based** microservices setups. It acknowledges that RPC processing constitutes a significant portion of communication latency, and simply optimizing the TCP/IP layer is insufficient for achieving faster RPCs. To address this, the paper advocates for the adoption of near-memory reconfigurable Network Interface Cards (NICs) to shift the entire RPC stack to hardware, thereby enhancing CPU utilization efficiency and reducing request latency. By harnessing hardware acceleration and leveraging memory interconnects, the Dagger framework endeavors to boost the performance of RPCs in cloud microservices environments while mitigating the shortcomings of traditional RPC frameworks.

Here are the detailed points outlining the proposed solution (Figure: 3.1):

1. Hardware-based Offloading: The paper suggests offloading the entire RPC stack to dedicated hardware, specifically near-memory reconfigurable Network Interface Cards (NICs). By moving the RPC processing to hardware, the solution aims to improve CPU efficiency and reduce request latency.

2. Integration with Memory Subsystem: Dagger integrates the FPGA-based reconfigurable RPC stack into the processor's memory subsystem over a NUMA interconnect. This integration allows for efficient communication between the hardware-accelerated RPC stack and the processor, leveraging memory interconnects instead of traditional PCIe protocols.

3. Software API: While the RPC stack is implemented in hardware, the software layer is responsible for providing the RPC API. This design principle allows for easy integration

of Dagger into existing applications and frameworks, enabling developers to leverage the benefits of hardware acceleration without significant changes to their codebase.

4. Configurability: Dagger offers modularity and configurability through SystemVerilog macros/parameters. Different hardware parameters and components can be selected, allowing for customization based on specific requirements. For example, the choice of CPU-NIC interfaces, transport layer, on-chip cache sizes, and flow FIFOs can be configured via hard configuration.

5. Multi-Tenancy and Co-Location Support: The proposed solution includes out-of-the-box support for multi-tenancy and co-location. A single physical FPGA adapter can host multiple independent NICs, serving different tenants running on the same host. This feature enhances resource utilization and enables efficient sharing of hardware resources among multiple microservices.

6. Performance Benefits: The paper demonstrates the effectiveness of Dagger by achieving significantly better per-core RPC throughput compared to previous solutions based on software RPC frameworks and specialized hardware adapters. The solution also shows performance improvements in terms of median and tail latencies when integrated with third-party applications like Memcached and MICA.

7. Virtualization: Dagger is designed to be virtualizable, allowing multiple instances of the NIC to be placed on the same FPGA. Each instance serves a dedicated microservice tier, enabling efficient resource sharing and fair round-robin sharing of the system bus and memory. This virtualization capability enhances scalability and flexibility in deploying microservice architectures.

8. Novel Approach: The proposed solution is the first attempt to leverage near-memory reconfigurable NICs closely coupled with processors over memory interconnects as programmable networking devices. This approach opens up opportunities to customize networking fabrics for frequently evolving interactive cloud services, addressing system challenges and optimizing performance.

The performance of the Dagger architecture (Figure: 3.2) has been extensively evaluated and compared with other baselines in the content. Dagger achieves significantly higher per-core RPC throughput compared to both highly optimized software stacks and systems using specialized RDMA adapters. It scales up to 84 Mrps with 8 threads on 4 CPU cores while maintaining state-of-the-art end-to-end latency. In comparison to other baselines, Dagger outperforms FaSST and the DPDK-based eRPC, showing 1.3 - 2.5x higher per-core RPC throughput. It also surpasses the performance of Memcached and MICA, achieving lower median and tail KVS access latency. For example, Dagger brings down the median KVS access latency to 2.8 - 3.5 us and

the tail latency to 5.4 - 7.8 us. Furthermore, Dagger demonstrates better CPU efficiency by offloading the entire RPC stack to hardware, resulting in higher per-core RPC throughput and lower request latency. It outperforms both optimized software RPC frameworks and specialized hardware adapters, reaching up to 12.4 - 16.5 Mrps of per-core throughput.

One of the limitations of the Dagger model is its reliance on hardware optimization. While this hardware-based solution offers significant performance improvements for RPC processing in cloud microservices, it also presents certain constraints.Firstly, the hardware optimization implemented in Dagger requires consistent deployment across all nodes in the cluster. This means that each node must have the same hardware configuration, including the near-memory reconfigurable Network Interface Cards (NICs). This uniformity ensures that the desired improvements in RPC processing can be achieved throughout the cluster. However, this requirement may pose challenges in terms of scalability and cost, especially for large-scale deployments where maintaining identical hardware configurations across all nodes can be complex and expensive. Secondly, the current design of Dagger lacks support for efficient RPC reassembling to handle larger requests. This limitation means that Dagger may not be suitable for scenarios where the size of the RPC requests exceeds its current capabilities. This can restrict the applicability of Dagger in certain use cases that involve large data transfers or complex RPC payloads.Furthermore, the availability of resources, such as the BRAM memory in FPGAs, can also pose limitations. While leveraging FPGA-managed on-chip memory improves the efficiency of NIC caches, the capacity of available memory may be limited. This limitation can impact the allocation of connection cache memory for NIC instances serving tenants with varying network provisioning requirements.It is important to note that these limitations are specific to the current design of Dagger and may be addressed in future iterations or alternative architectures. Nonetheless, understanding these limitations is crucial when considering the adoption of Dagger for specific use cases.

3.1.2 RPC Optimization across geo-distributed services

Geo-distributed services refer to a system architecture where services are deployed across multiple geographical regions. The main objective of geo-distribution is to bring services closer to users, reducing latency and improving performance. However, managing and optimizing the communication between these distributed services poses significant challenges. One of the main challenges in geo-distributed services is minimizing latency. As services are spread across different regions, the choice of routing requests becomes crucial. Balancing the trade-off between local queuing delay and cross-region network round-trip time (RTT) is a complex task. Simply minimizing RPC latency by sending requests to local servers can lead to overloading and high error rates when the queuing delay reaches a threshold. Another challenge is load balancing across

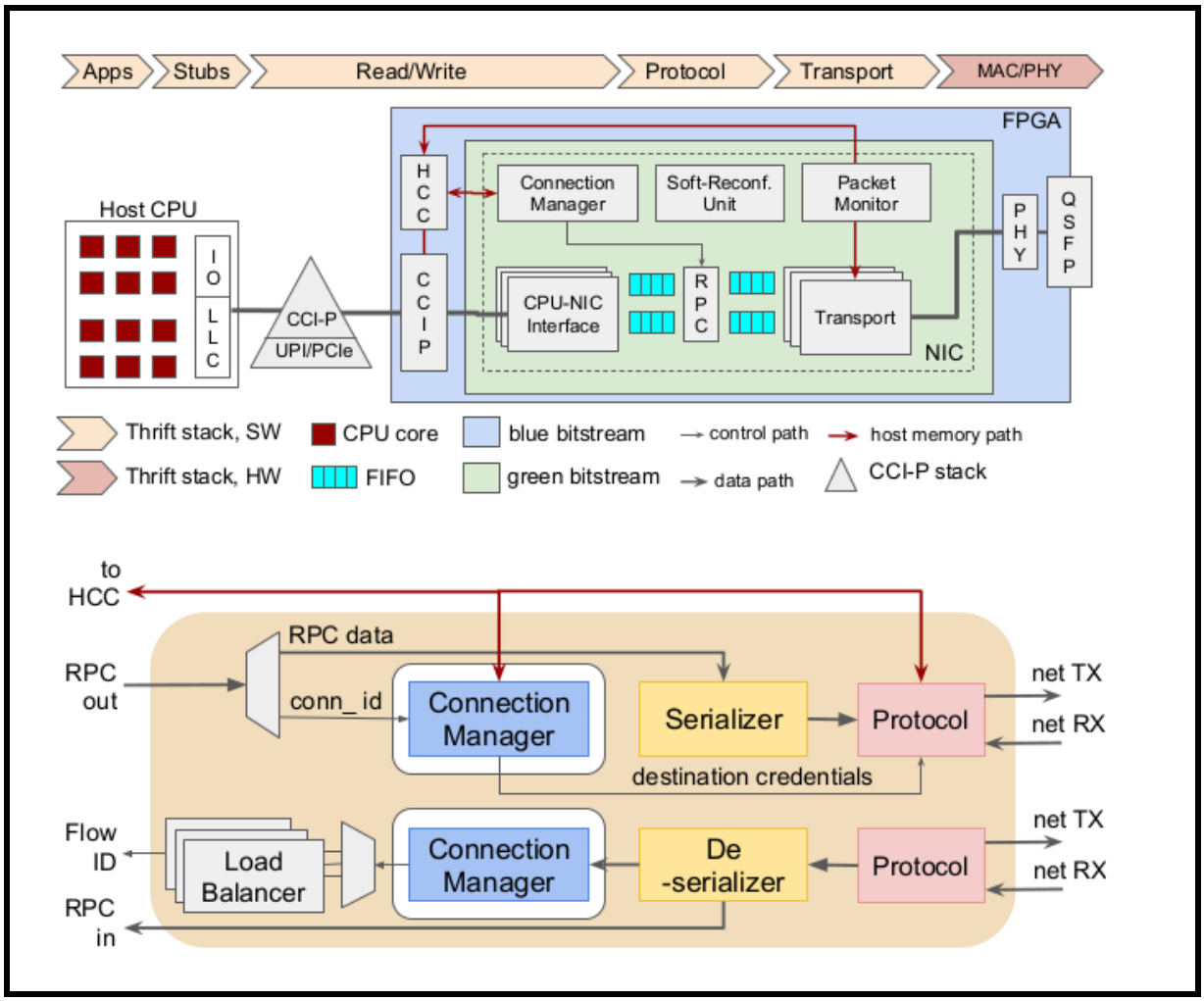

Figure 3.1: *Dagger: Proposed Architecture*

regions while considering both network RTT and server utilization. Predicting how traffic shifts would affect RPC latency at high utilization is not robust, making it difficult to optimize latency performance. Additionally, the need to consider regional locality in routing decisions adds complexity to the system.To address these challenges, service mesh architecture comes into play. A service mesh is an infrastructure layer that facilitates communication between microservices. It provides features like service discovery, load balancing, traffic routing, and observability. Each microservice is accompanied by a sidecar proxy, which handles the communication with other services. The service mesh acts as a centralized control plane, managing and configuring the sidecar proxies. By implementing a service mesh, developers can offload common networking concerns and improve the management of geo-distributed services. It enables efficient communication, scalability, resilience, and observability, ultimately enhancing the overall performance and user experience of the system.

Table 3: Median round trip time (RTT) and throughput of single-core RPCs compared to related work [1].

	IX	FaSST	eRPC	Net-DIMM	Dagger
Objects	64B msg	48B RPC	32B RPC	64B msg	64B RPC
TOR delay	N/A	0.3 us	0.3 us	0.1 us	0.3 us
RTT, us	11.4	2.8	2.3	2.2	2.1
Thr., Mrps	1.5	4.8 [2]	4.96 [2]	N/A	12.4

[1] Performance numbers are provided from corresponding papers
[2] Recorded in symmetric experimental settings

Figure 3.2: *Dagger: Experimentation results*

The importance of service mesh architecture lies in its ability to address the challenges faced in managing and optimizing communication between geo-distributed services. By providing features such as service discovery, load balancing, traffic routing, and observability, a service mesh enables efficient and reliable communication between microservices in a distributed system. It offloads common networking concerns from individual services, allowing developers to focus on building and deploying their services without worrying about the underlying infrastructure.Now, let me introduce the paper titled "**ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta**" [40]. This paper, presented at the 17th USENIX Symposium on Operating Systems Design and Implementation, discusses Meta's global service mesh called ServiceRouter (SR). The authors of the paper, Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang, present 11 years of work on SR, highlighting its scalability and cost-effectiveness.The paper focuses on the challenges of building a hyperscale service mesh and proposes novel techniques to optimize routing, minimize latency, and balance load across geo-distributed data center regions. It introduces the concept of locality rings to simultaneously minimize RPC latency and balance load, which is a unique approach not attempted before. The authors also compare different architectures of service mesh and discuss the design space and

components involved in SR. Overall, this paper provides valuable insights into the design and implementation of a global service mesh and offers practical solutions for building scalable and efficient service architectures. It is a valuable resource for those interested in understanding and implementing service mesh architectures in their systems.

This work is specific to Meta as it presents Meta's global service mesh called ServiceRouter (SR) (Figure: 3.3). The paper discusses the unique challenges faced by Meta in managing widely deployed libraries, such as SRLib, and proposes solutions to address these challenges. It highlights the importance of Configerator and Conveyor, powerful configuration management tools used at Meta, in managing not only SRLib but also other widely deployed libraries.

The paper proposes several solutions to address the latency problem in RPC (Remote Procedure Call) and communication in a geo-distributed service mesh. Here is a detailed explanation of the proposed solutions:

1. Latency-Focused Approach: The paper introduces a latency-focused approach to minimize RPC latency. This approach involves sending requests to local servers until the queuing delay reaches a threshold. However, the paper highlights the potential issue of severe overloading and high error rates when the queuing delay at local servers exceeds the threshold. To address this, the paper emphasizes the need for a robust method that considers both network RTT (Round-Trip Time) and server utilization to optimize latency performance.

2. Locality Awareness: The paper introduces the concept of locality rings to improve latency and load balancing. Each service can define a set of rings with increasing latencies. These rings filter out long-latency servers and samples from nearby servers to route requests efficiently. The Latency Monitoring Service (LMS) periodically updates RTTs between regions, and RPC clients obtain this information via the Centralized Monitoring Service (CMS).

3. Load Balancing and Traffic Routing: The paper highlights the importance of load balancing across regions while considering network RTT and server utilization. It discusses the limitations of predicting how traffic shifts would affect RPC latency at high utilization. To address this, the paper proposes an adaptive approach to load estimation based on workload characteristics. This approach allows for more accurate prediction and optimization of RPC latency.

4. RPC Connection Reuse: The paper addresses the overhead associated with setting up new TLS/TCP connections for each RPC request. It discusses the inefficiency of connection reuse with random sampling techniques. To maximize efficiency, the paper proposes sampling two random servers from a stable subset of servers, rather than all servers. This

approach aims to maximize connection reuse and reduce the overhead of establishing new connections for each RPC request.

The paper presents experimental results to demonstrate the effectiveness of the proposed solutions in optimizing latency in RPC and communication. The experiments were conducted using different payload sizes and compared the performance of SRLib and SRProxy with Thrift. The results showed that using production-sized payloads, SRLib and SRProxy consumed 80 % and 273 % additional CPU cycles, respectively, compared to Thrift. Additionally, the experiments showed that load balancing was effective in balancing the load across servers, with low coefficient of variation (CV) for unsharded services. These results indicate that the proposed techniques can significantly improve latency performance in a distributed service mesh.

Despite the positive results, the paper acknowledges certain limitations of the proposed solutions. One limitation is the high overhead associated with SRProxy, which would require a significant number of additional machines. Additionally, the paper highlights the inability of the proxy to perform zero-copy data forwarding due to the need for encryption and identity management. Another limitation is the challenge of accurately predicting how traffic shifts would affect RPC latency at high utilization. These limitations suggest that further research and optimization may be required to address these challenges and improve the overall effectiveness of the proposed solutions.

RPC optimization is crucial for achieving better performance in microservices. In a distributed system where microservices communicate with each other through remote procedure calls (RPCs), minimizing RPC latency is essential. By optimizing RPCs, we can significantly improve the overall performance and efficiency of microservices. This includes reducing network Round-Trip Time (RTT), balancing load across geo-distributed datacenter regions, and optimizing cross-region routing. These optimizations ensure that RPC requests are processed quickly and efficiently, resulting in faster response times, improved scalability, and enhanced user experience. By implementing RPC optimization techniques, microservices can achieve better performance, increased throughput, and seamless communication, ultimately leading to more efficient and reliable systems.

3.2 Tail Latency

In the realm of microservices, tail latency emerges as a crucial metric, representing the outliers in response times experienced by a system. Unlike average latency, which provides a general overview of performance, tail latency focuses on the extreme cases where responses take significantly longer than the norm. These delays, often caused by various factors such as resource contention, inefficient algorithms, or unexpected spikes in demand, can undermine user experience and system reliability.

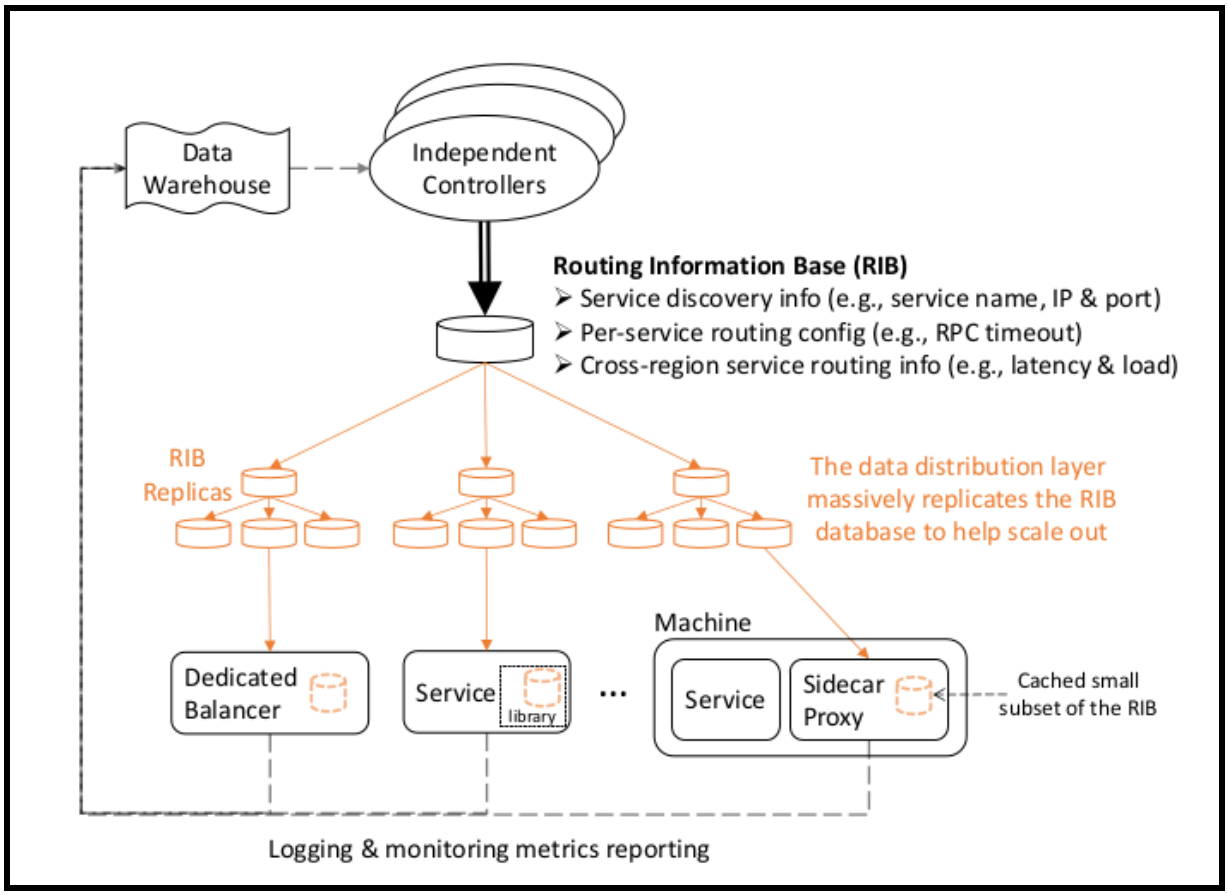

Figure 3.3: *ServiceRouter*

Tail latency in microservices specifically pertains to the variability in response times exhibited by individual service endpoints within a distributed architecture. Each microservice, responsible for specific functionalities, might encounter distinct bottlenecks or performance hurdles, leading to sporadic delays in processing requests.

Comparatively, latency stemming from network or inter-process communication relates to the time taken for data to travel between different components or services within the microservices ecosystem. While network latency contributes to overall response times, it differs from tail latency in that it typically represents the consistent overhead incurred by transmitting data across network boundaries. In contrast, tail latency highlights the irregular delays that occur within individual service interactions, often reflecting underlying issues within the microservice itself rather than the network infrastructure.

Optimizing tail latency stands as a paramount objective in the pursuit of efficient microservices architecture. Unlike average latency, which offers a generalized view of system performance, tail latency represents the outliers that can significantly impact user experience and system reliability. In a microservices environment, where services operate independently and interact

extensively, even a few instances of prolonged response times can cascade into widespread disruptions.

Efficient microservices rely on consistent and predictable performance to meet stringent user expectations and service-level agreements (SLAs). High tail latency can undermine these goals, leading to user frustration, decreased engagement, and potential business losses. Furthermore, in scenarios where microservices orchestrate complex workflows or handle critical transactions, optimizing tail latency becomes imperative to ensure smooth operation and maintain data integrity.

Moreover, in distributed systems characterized by microservices, latency variability can exacerbate resource contention issues, leading to inefficient resource utilization and increased operational costs. By minimizing tail latency, organizations can enhance resource efficiency, enabling them to scale their microservices architecture effectively while containing infrastructure expenses.

Furthermore, optimizing tail latency fosters a more resilient and fault-tolerant microservices ecosystem. By identifying and addressing the root causes of latency outliers, such as inefficient algorithms, database queries, or network bottlenecks, organizations can fortify their systems against potential failures and mitigate the risk of service degradation or downtime.

3.2.1 Optimizing Scheduling strategies

The paper "Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling" [41] highlights the challenge of achieving microsecond-scale tail latency in data center applications. Tail latency refers to the response time of the slowest request in these applications, such as web search, e-commerce, and social networking. These applications have strict service level objectives (SLOs) for tail latency, meaning they aim to provide fast and responsive user experiences. The paper highlights that existing systems face significant overheads at both current and future timescales, which hinder their ability to meet the SLOs for microsecond-scale tail latency. The overheads arise from various factors, including preemptive scheduling, inter-thread communication, and the dedicated dispatcher that runs no application logic.

The paper proposes a scheduling runtime called Concord (Figure: 3.4) to address the challenge of achieving microsecond-scale tail latency in data center applications. Here are the key solutions presented in the paper:

1. System Model: The paper introduces a system model that includes a dedicated dispatcher thread and multiple worker threads pinned to individual CPU cores. This model forms the basis for analyzing the trade-offs of different implementations.

2. Throughput Overheads Analysis: The paper analyzes the throughput overheads that arise from preemptive and single queue scheduling at the microsecond scale. It identifies three

main sources of overhead: preemptive scheduling, inter-thread communication, and the dedicated dispatcher that runs no application logic.

3. Approximation of Optimal Scheduling: Concord's design is driven by the insight that careful approximation of optimal scheduling policies enables efficient, low-overhead mechanisms. By approximating optimal scheduling, Concord aims to strike a balance between tail latency and throughput.

4. Benefits of Concord: The paper evaluates Concord's performance and demonstrates its throughput benefits. It shows that Concord can significantly improve application throughput while maintaining low tail latency costs. It achieves this by introducing new techniques to optimize existing scheduling policies.

The proposed algorithm demonstrates promising results in addressing the challenge of achieving microsecond-scale tail latency in data center applications. The algorithm, implemented in the Concord scheduling runtime, aims to optimize scheduling policies and minimize throughput overheads while maintaining low tail latency costs. Through extensive evaluation and analysis, the paper shows that Concord performs well across different service time distributions and real latency-sensitive applications. It compares Concord's performance with existing systems and baselines, highlighting its significant improvements in reducing overheads and enhancing application throughput. The algorithm's effectiveness is demonstrated through various metrics and experiments, including comparing the throughputs sustained by Concord and other systems, evaluating the overhead and timeliness of Concord's instrumentation, and measuring the benefits of its mechanisms. The results indicate that Concord achieves notable reductions in overheads and provides efficient mechanisms for improving application performance.

The algorithm has a few limitations that are worth considering. Firstly, it requires the availability of application source code written in a compiled language with an LLVM backend. LLVM stands for Low-Level Virtual Machine. It is a compiler infrastructure that provides a collection of modular and reusable compiler and toolchain technologies. LLVM is designed to optimize program performance, enable efficient code generation, and support various programming languages. While this may not be an issue for developers deploying Concord on bare metal or for tenants deploying their low-latency systems on VMs in the public cloud, it poses challenges for using Concord as a runtime provided by public cloud providers. This is because it would require tenants to share their code with the cloud provider, which may not be desirable in certain scenarios. Secondly, the current Concord prototype is restricted to single-dispatcher systems. While this limitation may not be a concern for low CPU count environments, such as small VMs, it can become a bottleneck as the number of CPUs increases and the service time decreases. In such cases, replication or trading off throughput for tail latency can be explored as potential solutions to improve scalability.

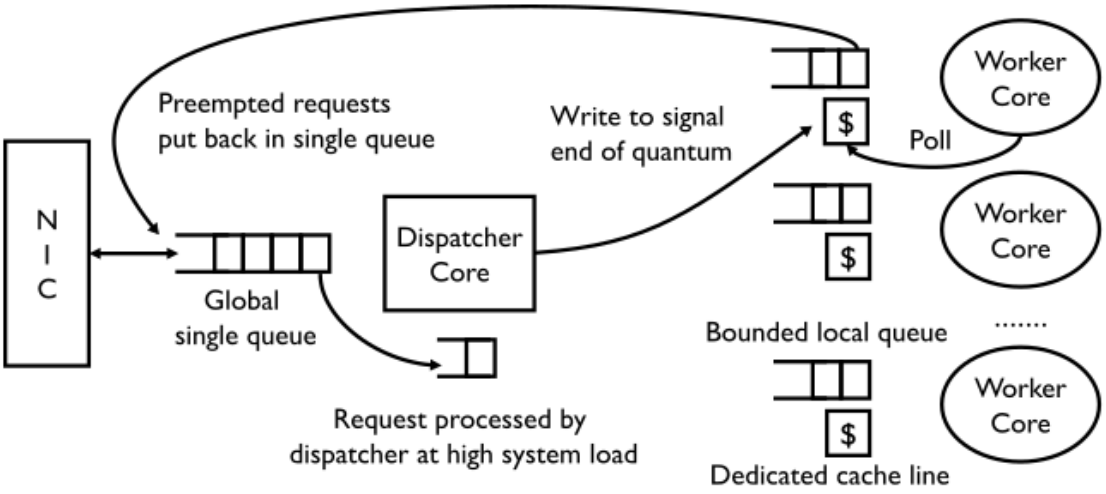

Figure 4. The Concord architecture. Concord's compiler-enforced cooperation relies on communicating via a dedicated cache line, JBSQ(k) employs bounded core-local queues to eliminate coherence stalls, and the Concord dispatcher steals work at high load to contribute to application throughput.

Figure 3.4: *Proposed Scheduling Algorithm: Concord*

3.2.2 Configuration Optimization

The paper "Towards Optimal Configuration of Microservices" [42] highlights the novel challenge of tuning the configuration parameters of microservices applications to minimize tail latency. Microservices architecture allows for the decomposition of applications into smaller, independent modules, but this modularity leads to a large configuration space with potentially interdependent parameters. The paper aims to investigate various optimization algorithms and techniques to effectively tune these configuration parameters and reduce tail latency. By conducting extensive experimental investigations, the paper aims to identify the best optimization algorithm that can significantly improve the tail latency of microservices applications. The goal is to find the optimal configuration settings that can minimize tail latency and enhance the overall performance and responsiveness of microservices applications.

The paper evaluates different optimization algorithms, including DDS (Dimensionality Decreasing Search), PSO (Particle Swarm Optimization), and BO (Bayesian Optimization). These algorithms are applied to tune the configuration parameters of microservices applications. The

authors compare the improvements in tail latency achieved by each algorithm and analyze their convergence and overhead. This evaluation provides insights into the effectiveness and efficiency of different optimization techniques in minimizing tail latency and optimizing the performance of microservices applications.

1. DDS (Dimensionality Decreasing Search): dimensionality reduction approaches involve identifying a subset of microservices that have the most impact on end-to-end application latency. By reducing the dimensionality, the authors aim to achieve similar improvements in tail latency while tuning only a fraction of the total microservices. This approach allows for more efficient optimization of the configuration parameters, leading to enhanced performance and responsiveness of microservices applications.

2. PSO (Particle Swarm Optimization): PSO is an evolutionary algorithm inspired by the behavior of bird flocking or fish schooling. It works by iteratively improving candidate solutions, known as particles, based on the best value each particle has seen so far (exploration) and the global best value seen by the entire swarm (exploitation). PSO moves particles around the search space to find the optimal configuration settings that minimize tail latency. The paper compares the improvements in tail latency achieved by PSO with other optimization algorithms.

3. BO (Bayesian Optimization): BO is a sequential model-based optimization algorithm that approximates the objective function. It uses a probabilistic model to capture the relationship between the configuration parameters and the tail latency. BO iteratively selects new configurations to evaluate based on the information gained from previous evaluations. By leveraging the probabilistic model, BO aims to find the globally optimal or locally optimal configurations that minimize tail latency. The paper evaluates the performance of BO in comparison to other optimization algorithms.

The paper evaluates different optimization strategies (Figure: 3.5) to determine which one works best for minimizing tail latency in microservices applications. The authors compare the performance of three optimization algorithms: DDS (Dimensionality Decreasing Search), PSO (Particle Swarm Optimization), and BO (Bayesian Optimization). The evaluation results show that DDS provides the best improvement in tail latency, with a 23.4% reduction compared to the default configuration. PSO and BO also perform well, achieving improvements of 22.9% and 22.1% respectively. The authors also discuss the improvements in tail latency compared to the configuration employed by the DeathStarBench benchmark developers, which showed a 3% improvement. This highlights the effectiveness of the optimization algorithms proposed in the paper. Furthermore, the authors address the challenge of a large configuration space by employing dimensionality reduction techniques. By reducing the dimensionality, they can achieve

similar improvements in tail latency while tuning only a fraction of the total microservices. This approach helps in minimizing the overhead of optimization and streamlining the tuning process.

However, this approach to optimizing the configuration parameters of microservices applications has certain limitations. One limitation is the reliance on dimensionality reduction techniques to identify a subset of microservices that have the most impact on end-to-end application latency. While this approach helps in reducing the configuration space and streamlining the optimization process, it may overlook certain microservices that could potentially contribute to tail latency. The effectiveness of dimensionality reduction techniques heavily depends on the accuracy of the chosen subset and the assumptions made during the reduction process. Another limitation is the evaluation of a limited number of optimization algorithms, namely DDS, PSO, and BO. While these algorithms have shown promising results in minimizing tail latency, there may be other optimization algorithms that could potentially outperform them. The authors' approach could benefit from a more comprehensive evaluation of a wider range of optimization algorithms to ensure the selection of the most effective technique. Additionally, the authors' approach assumes that the identified critical microservices and the chosen optimization algorithms will remain consistent over time. However, the performance characteristics of microservices applications can change dynamically due to various factors such as workload variations, system updates, or changes in user behavior. Therefore, the authors' approach may require regular re-evaluation and adaptation to ensure its continued effectiveness.

Despite these limitations, the authors' approach provides valuable insights and a foundation for optimizing the configuration parameters of microservices applications. By addressing these limitations and considering future research directions, the approach can be further enhanced to achieve even better performance and responsiveness in microservices applications.

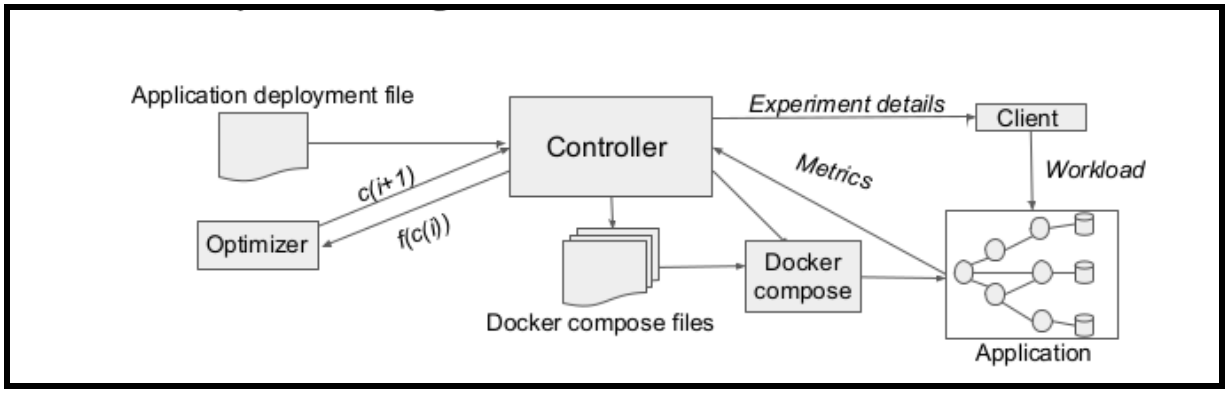

Figure 3.5: *Proposed Algorithm*

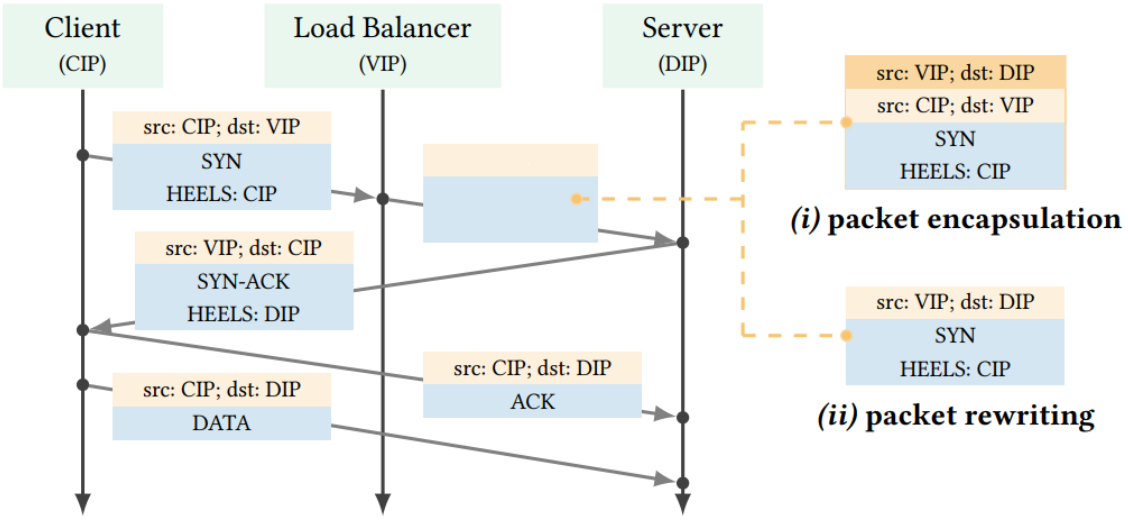

Figure 3.6: *TCP sequence diagram of HEELS*

3.3 Load Balancing Architectures

Section 2.4 comprehensively details all the challenges we must consider while designing a load balancer for a microservice architecture. In general, load balancers are tasked for dividing traffic to a group of distributed servers, often with the added responsibility of ensuring security and fault tolerance of the overall service maintaining minimal service latencies and operational overheads. In this section, we explore a few load balancer architectures for microservices from both - a research and an industrial perspective, the performance issues they aim to solve along with their shortcomings and scope for improvement.

3.3.1 Host Enabled Ebpf based Load balancing Scheme [4]

The paper first weighs the pros and cons of centralized and decentralized load balancing schemes at the Network Layer (L4). Centralized schemes with a single load balancer are more efficient in terms of algorithmic complexity and scheduling / routing policies, at the cost of heavy computation cost at a single node, making it a single point of failure. Their decentralized counterparts do not face these problems and allow better horizontal scaling, but introduce new ones: communication and synchronization amongst the load balancers and the overhead of a more difficult management scheme. HEELS is a novel architecture that combines the best of both types: centralized (dedicated middlebox acting as load balancer) and decentralized (network of servers, each tasked with balancing load) designs for cloud computing and elastic microservices.

Figure **??** demonstrates the architecture of HEELS. HEELS uses a centralized load balancer receiving requests from clients to be mapped to a group of backend servers. HEELS utilizes

a customized TCP option called *HEELS* in all TCP packets. The client sets this to Client IP (*CIP*) while sending the request to the load balancer's Virtual IP (*VIP*). The load balancer rewrites the source IP from *CIP* to *VIP* for normal packet redirection, while maintaining *HEELS=CIP* before forwarding it to a chosen destination server (*DIP*). Now, upon receiving the packet, the server can set the destination to *CIP*, so as to directly connect with the client and set source as *VIP* (of load balancer) so the client believes the packet comes from the load balancer, and send its own IP (*DIP*) as the option *HEELS=DIP*. At the client-side, the option *HEELS* provides the destination server. This scheme does not require any kernel modifications or insertion of kernel modules at the client, load balancer or server at all, since the source and destination addresses are set appropriately at every stage to use the regular TCP protocol. In addition, the load balancer comes into play only once while establishing the TCP handshake and all further communications are directly between the server and client, thereby decentralizing all other requests.
HEELS is implemented using eBPF programs to work with the custom TCP option *HEELS*. It is also a architecture-agnostic general solution to effectively decentralize any centralized load balancing scheme, making it deployable to public cloud systems and compatible with proprietary load balancers. Experimental results show that HEELS adds no data transfer latency and has a very small throughput overhead of $\tilde{3}\%$.

3.3.2 Uber's Cinnamon [5]

Uber is one the largest online transportation companies, providing services such as travel and food delivery to an average of >100M customers on a monthly basis. Uber has thousands of microservices at its backend, serving millions of requests per second. In this section, we explore an interesting approach of "Load Shedding" used by Uber's microservice architecture, which effectively rejects a fraction of requests in situations of extreme load. Uber currently uses a load shedder library Cinnamon, to perform automatic graceful degradation whose goal is to provide the customers a seamless user experience even if parts of the backend are overloaded. Figure **??** presents the traversal of a request through Cinnamon before reaching the business logic (in the backend servers)

Cinnamon defines the priority of each user request as a tuple of attributes <*tier, cohort*>, where <*tier*> defines the importance of a request (range: 0 (highest) - 6 (lowest)) and <*cohort*> (range: 0 - 127) defines the set of user requests to shed ensuring that all microservices drop the requests of the same set of users. With 6x128 = 768 priorities, Cinnamon utilizes an efficient bucket-based priority queue, as in **??** to sort the importance of requests, which schedules higher priority requests before lower priority ones. The Rejector component in **??** detects whether a particular endpoint is overloaded and sheds some fraction of the requests to manage

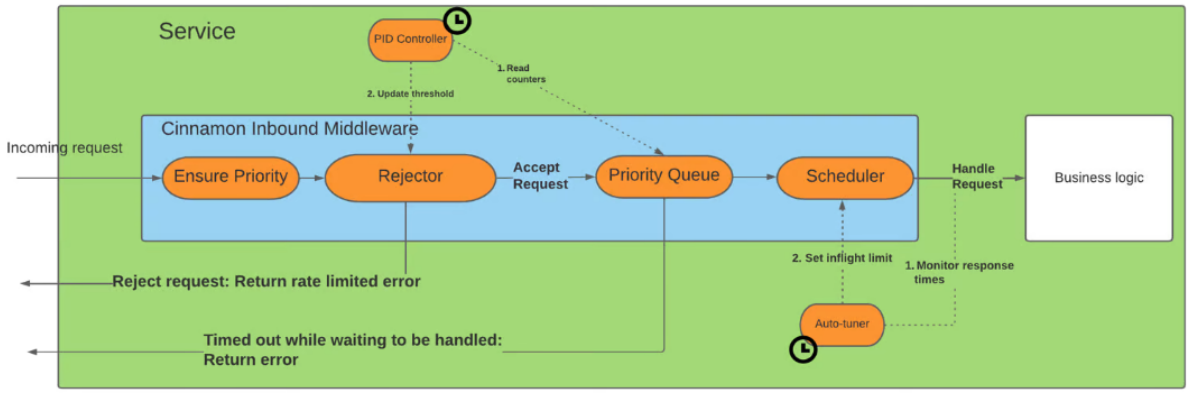

Figure 3.7: *Lifecyle of a request in Cinnamon*

the size of the request queue. If the reqeust queue for an endpoint has not been emptied even once within a predefined timeout, then the endpoint is declared as being overloaded. Upon overloading, an efficient PID* controller determines the ratio of load to shed, ensuring that this ratio increases only gradually and does not suddenly drop lots of user requests.
Finally, the scheduler component dispatches requests accepted for processing to the backend logic, while limiting the number of concurrent requests allowed to a maximum threshold to control the latencies. This threshold of concurrency is automatically tuned by an auto-tuner using a modified version of the Vegas TCP/IP congestion control algorithm [43]. *A Proportional–Integral–Derivative (PID) controller is a feedback driven control loop mechanism.
Experimental results show that Cinnamon can preserve good latencies while servicing bigger overloads and adds only 1 microsecond of overhead per request, meaning it is very efficient. Figure **??** shows the throughput of Cinnamon compared to Qalm (Uber's older load shedder) at a request rate of 4000 requests per second. The first bump (with throughput = 700 req/s) is the baseline with no load shredder.

3.3.3 Netflix's Ribbon [6]

Netflix is one of the most popular online streaming platforms with over 260 million active subscribers globally. Netflix has adopted a fine-grained service oriented architecture with hunders of microservices. User requests to these services are handled by a collection of a few "Edge Services" communicating with each other using a lightweigh REST based protocol. In this section, we discuss Netflix's Ribbon, which is used for this interprocess communication for east-west traffic (services internally talking with each other). Note that for north-south traffic (mapping user requests to services), Netflix uses Amazon's Elastic Load Balancing (ELB).

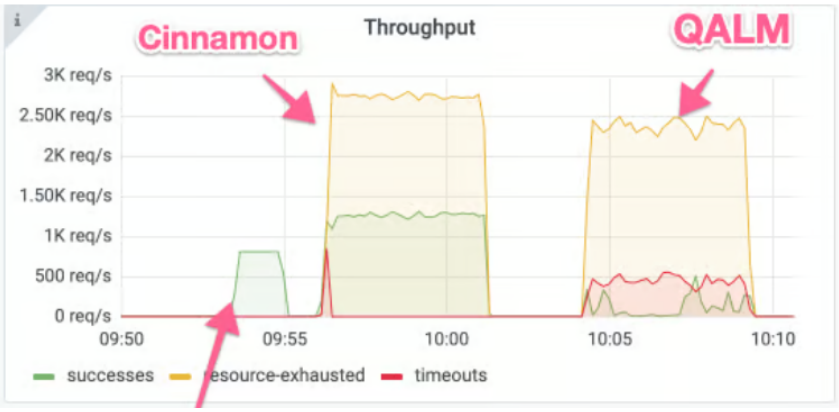

Figure 3.8: *Throughputs of Cinnamon and Qalm at a load of 4000 requests/s*

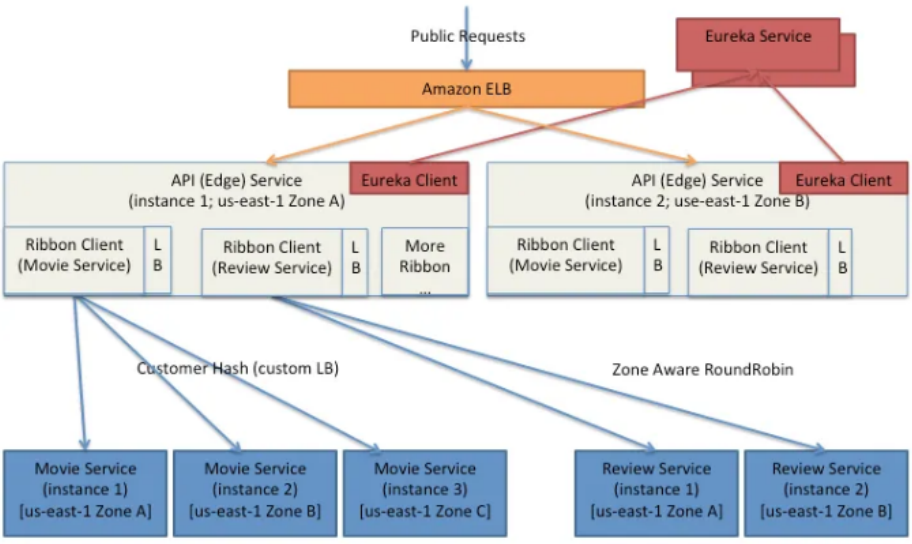

Figure 3.9: *Microservice Deployment Architecture at Netflix*

Figure 3.9 depicts the deployment architecture used by Netflix for balancing load to its microservices. Ribbon, available as an open source project, focuses on providing pluggable load-balancing algorithms for intercommunication amongst Netflix's microservices. Some of the customizable strategies provided are:

- Simple Round Robin
- Weighted Response Time
- Random Round Robin
- Zone Aware Round Robin

The first three techniques are straightforward and self-explanatory. We elaborate on the fourth technique - Zone Aware Round Robin. As the name suggests, it is configured to choose a service hosted in a zone geographically close to the source where the request originates, thereby reducing the network latency. When a client request is received, the load balancer first analyzes the statistics at all the available zones. If any the load at any zone has crossed a given threshold, it is dropped from the list. From the remaining zones, another factor to consider is the number of available instances there, so the probability of choosing a zone is bumped up by the number of live instances. Finally, a server is chosen from this zone using one of the Load Balancing strategies listed above. The source code for Ribbon can be found at `https://github.com/netflix/ribbon`

3.4 Resource Allocation, Consistency and Concurrency

One of the biggest challenges in microservices, and distributed systems in general is resource sharing and data consistency among replicas. These issues become more critical in light of microservices that are clusters of thousands if not hundreds of applications running concurrently, while being distributed over the network. It is very challenging to ensure the well-known ACID properties of database transactions: Atomicity, Consistency, Isolation and Durability in a microservice framework using a database-per-service pattern, wherein each replica maintains a copy of its own data. There are two main approaches to achieving these goals in a distributed system:

1. Distributed Transactions
2. Eventual Consistency

We now discuss both legacy traditional methods and novel improvements upon these methods made specifically for microservices for these two techniques.

3.4.1 Distributed Transactions

Distributed transactions guarantee strong consistency and support the ACID features same as in a local database system. The famous 2-Phase Commit (2PC) protocol performs distributed transactions. The basic idea is that one of the replicas acts as the co-ordinator for other replicas. Upon a write request, it forwards the request to all replicas and waits for their acknowledgements. The replicas remember the transaction in a fail-safe way, possibly a write-ahead-log but do not yet execute it. If all replicas acknowledge this, the co-ordinator now sends a request to commit this transaction and also commits it itself. As is obvious, a major issue with this protocol is that it is blocking and requires locking the resource to be written. with the possibility of deadlocks. Also, the transaction co-ordinator is a single point of failure. [44] presents 2PC*, an optimization attempt over the traditional 2PC protocol. 2PC* divides each lock into a two-level lock, which they call a Secondary Asynchronous Optimistic Lock (SAOL). This lock is taken asynchronously, thereby making 2PC* a non-blocking and deadlock-free algorithm. In addition to this, 2PC* also improves upon 2PC's concurrency protocol to reduce the probability of conflict between concurrent transactions using a graph-based approach involving strongly connected components computed using Tarjan's Algorithm. To achieve fault tolerance, 2PC* persists the transaction logs in each replica to the disk with a Paxos-based replication protocol for log data synchronization across multiple machines. The paper provides formal algorithms and proofs of correctness of the same. Their extensive evaluation results show an increase in throughput by atmost 3.3 times and reduction in latency by 67% under situations of heavy contention and high transaction load as compared to the traditional 2PC algorithm.

3.4.2 Eventual Consistency

Eventual Consistency follows a fundamentally different model as compared to Distributed Transactions. It does not guarantee the ACID properties, but rather provides BASE properties, which are:

1. **B**asically **A**vailable: Having high availability of data, non-blocking
2. **S**oft - state: System state may change without external input
3. **E**ventual consistency: Consistency is guaranteed eventually, not immediately

So, ACID properties prefer safety over liveness, whereas BASE properties prefer liveness over safety. SAGA is a common database pattern that operates on the BASE model. SAGA is an asynchronous model, where steps underlying the distributed transaction are performed by different replicas (microservices in our case) separately. A multistep distributed transaction can be modeled as a saga or series of local events at different replicas. Each replica, after executing

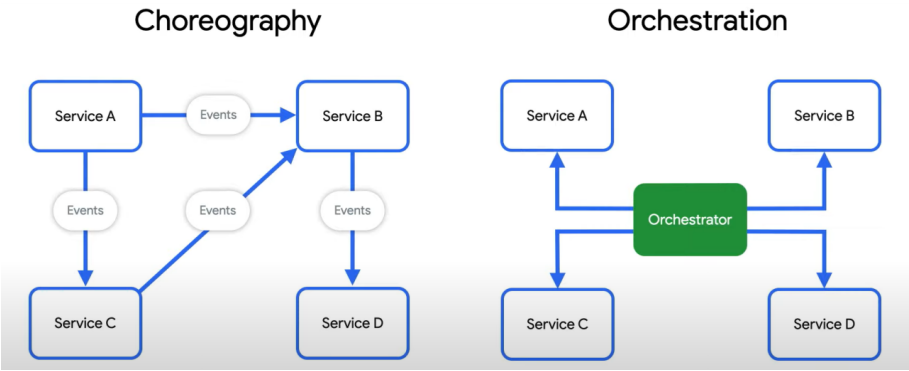

Figure 3.10: *SAGA Models*

its own steps, emits an event which triggers the next step, possibly at a new replica. In case of failure of any step, the event signals the previous steps to be rolled back at the older replicas. There are two ways of implementing SAGA 3.10:

1. Choreography-Based Saga: Events published by local transactions trigger the next transactions in other services

2. Orchestration-Based Saga: A central orchestrator object commands the services to execute certain transactions

3.4.3 Resource Management in Microservices

We have explored two ways of executing transactions across microservices 3.4.1 3.4.2. Such transactions can often operate not only on databases local to the server instances, but also resources shared across microservices. Resource management in frameworks with hundreds and thousands of microservices is a very difficult task. Allocation and access to shared resources in such an environment directly affects performance metrics such as latency and throughput. ERMS [45] proposes a novel approach to improve shared resource utilization in microservices while respecting service-level agreements among the different services. Figure 3.11 presents the system architecture of ERMS. The *tracing co-ordinator* generates dependency graphs for resources shared between the microservices and calculates the latencies of individual microservices from historical traces. The *offline profiling* module receives these latencies and accodingly chooses a model to profile the tail latency of microservices as a piece-wise linear function of the workload. The *graph merge* component simplifies the complex dependency graph into a simpler graph with only sequential dependencies for easier computation of target latencies by the *latency target computation* component. The *priority scheduler* assigns each microservice a different priority for accessing some resource based on its target latency. Allowing access to the shared resource based on this priority, in turn dynamically shifts the workload of the system.

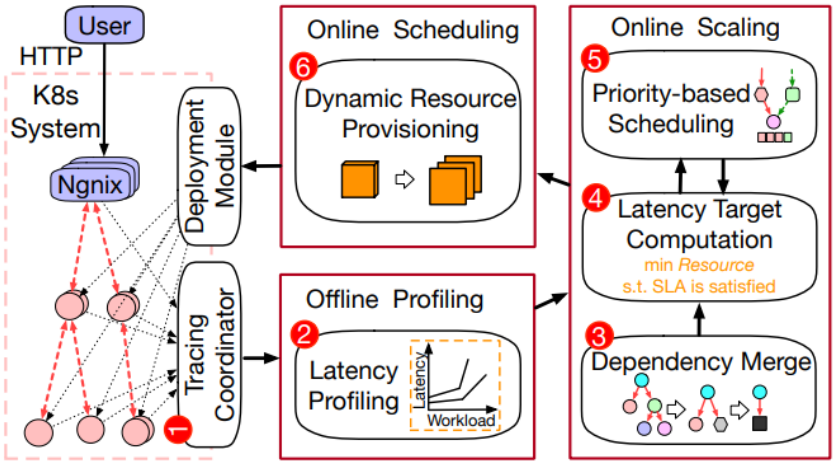

Figure 3.11: *System Architecture of ERMS*

The *dynamic resource provisioning* module detects such changes and manages the count of microservice replicas by scaling up or down. This process runs continuously to guarantee SLAs among the services while incurring minimal overhead and maintaining good latency values.

3.5 Monitoring Proprietary Cloud Microservices

With the ever-increasing demand for computation worldwide, there has been an industry shift from adopting standalone monolithic services to microservice architectures for the same. Cloud computing services like Google Cloud, Amazon AWS and Microsoft Azure are being increasingly adopted for various latency-critical interactive services. This shift towards microservice architectures has brought forth new challenges in terms of monitoring these distributed systems, primarily due to the increased complexity and dynamic nature of microservices. However, it is little empirically known how the software industry designs, monitors, and tests microservices systems.

Multiple representative, open-source benchmarks have been proposed as a means to approximately evaluate the performance metrics of proprietary microservices[46, 47]. However, benchmarking closed-source large-scale cloud microservices accurately is not straightforward - the representative, open-source benchmarks are unable to capture the full complexity of proprietary microservices.

There are three approaches to performing studies that require cloud applications:

(i) ***Using real services***: Even though leveraging real production services [48, 3] should

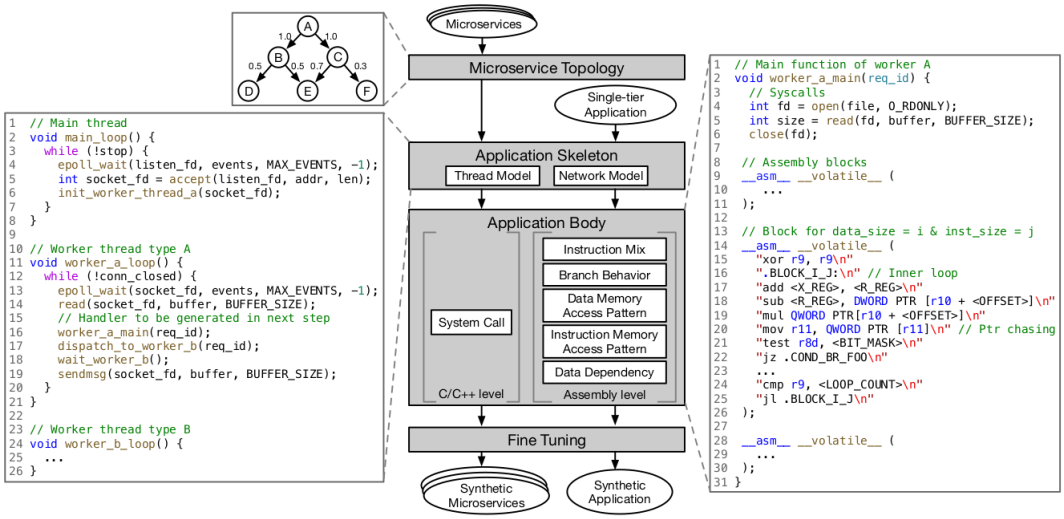

Figure 3.12: *Overview of the Ditto Synthetic Benchmark Generation framework.*

provide the most representation, such production services are rarely publicly available, and open-source applications, although useful, often lack the complexity of a real-world deployment.

(ii) ***Using simulation or trace replay***: Microarchitectural simulators, including gem5 [49] and ZSim [50] can accurately simulate the CPU performance of a given binary. However, it has been shown that studies that rely on simulation or replaying traces from a production system are tied to the system configuration the trace was collected on, and cannot easily generalize to arbitrary studies.

(iii) ***Generating representative synthetic services***: Synthetic benchmarks like NanoBench [51] and MicroGrad [47] offer a middle ground, with the synthetic application capturing critical features of the original service, but being malleable enough to adjust to different studies. However, cloud applications largely operate at the kernel level, and the above approaches only measure low-level performance metrics like IPC, cache miss rate and dependency distance, while leaving out the more relevant higher-level metrics like Tail Latency (Discussed in Section 3.2).

To this end, the paper [52] proposed Ditto, which is a synthetic benchmark generation framework that profiles an application at runtime and extracts key performance metrics using dynamic emulators (SystemTap, Valgrind, eBPF and Intel SDE). Then, it generates a synthetic service which preserves the performance of the original, using an entirely distinct code sequence, to avoid revealing the implementation of the original service.

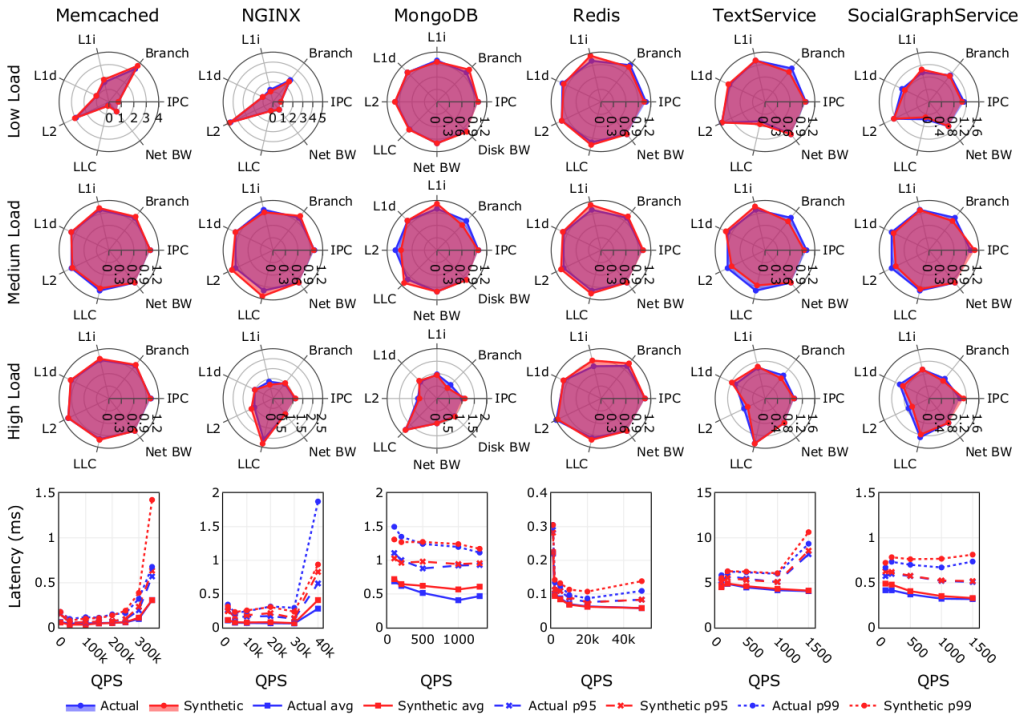

Figure 3.13: *PU performance metrics (IPC, branch mispredictions, L1i, L1d, L2 and LLC miss rates), network bandwidth, disk bandwidth (MongoDB only) and service latency under varying load across six services.*

Figure 3.12 shows an overview of Ditto's generation process. Ditto first learns their Remote Procedure Call (RPC) dependency graph using distributed tracing **(Microservices Topology)**. This graph is then used to create the API interfaces between the different synthetic microservices. Next, Ditto analyzes the thread and networking model, e.g., single- or multi-threaded, and synchronous or asynchronous respectively using kernel-level profiling, and builds the skeleton of each service **(Application Skeleton)**. The application skeleton contains empty handlers which are filled with appropriate functionality in the next step. The handlers can either be triggered upon receiving requests for worker threads, or by a timer for background threads. To generate the synthetic application body **(Application Body)**, Ditto instruments the application binary using kernel and user-space profilers for different subsystems. Finally, Ditto uses the deviation in performance metrics between original and synthetic application to fine tune the generator **(Fine Tuning)**.

The authors conduct an evaluation of Ditto's benchmarking capabilities on a on a heterogeneous cluster over six workload generators: **Memcached**, **NGINX**, **MongoDB**, **Redis** and **Social Network** (comprised of two tiers **TextService** and **SocialGraphService**). For all synthetic applications, they use the same load generator as the original application, sending dummy requests with the same traffic distribution.

Figure 3.13 shows CPU, network and disk performance metrics, and latency for six applications un- der different QPS. All applications are generated using profiling data under medium load. The upper three rows show IPC, branch misprediction, L1i, L1d, L2, LLC miss rates, and network and disk I/O bandwidth under low, medium, and high load, with relatively low average errors across all applications. This indicates that Ditto accurately clones the overall hardware performance metrics. Also, applications can have very different characteristics under different loads are accurately captured by Ditto in their synthetic counterparts.

The advantages of Ditto over previously proposed open-source representative synthetic benchmark generators can be summarised by the following design principles followed by Ditto:

(i) **End-to-end system stack modeling**: Ditto captures the inputs, RPC dependency graph, aplication binary, OS kernel, CPU, memory and resource interference.

(ii) **Portability**: Ditto uses platform-independent features to make sure that generated services are portable across platforms without having to re-profile them.

(iii) **Abstraction**: Ditto does not disclose the implementation of the original applica-

tion, only exposing the skeleton and post-processed performance characteristics to the synthetic benchmark user. Thus, the synthetic workload can be publicly shared, without a user reverse engineering the implementation of the original service.

(iv) **Automation**: Ditto automates the profiling and generation process. Users are not required to have expertise in the implementation of a service to use the framework.

References

[1] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 482–497, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Akshitha Sriraman and Thomas F. Wenisch. suite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[3] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Rui Yang and Marios Kogias. Heels: A host-enabled ebpf-based load balancing scheme. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 77–83, 2023.

[5] J. H. Thomsen et. al. Uber cinnamon. Technical report, Uber, 2023. `https://www.uber.com/blog/cinnamon-using-century-old-tech-to-build-a-mean-load-shedder/`.

[6] Sudhir Tonse Allen Wang. Netflix ribbon. Technical report, Netflix, 2023. `https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62`.

[7] Luiz André Barroso and Urs Hoelzle. The Datacenter as a Computer: An introduction to the design of Warehouse-Scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 1 2009.

[8] Christina Delimitrou and Christos Kozyrakis. QoS-Aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems*, 31(4):1–34, 12 2013.

[9] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James W. Lewis, and Stefan Tilkov. Microservices: the journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 5 2018.

[10] David E. Sprott. *Understanding Service-Oriented architecture*. 7 2006.

[11] Krzysztof Rakowski. *Learning Apache Thrift*. 12 2015.

[12] Jason Flinn. Cyber Foraging: bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 9 2012.

[13] Ragib Hasan, Md. Mahmud Hossain, and Rasib Khan. Aura: An iot based cloud infrastructure for localized mobile computation outsourcing. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 183–188, 2015.

[14] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909, 2018.

[15] Victor Velepucha and Pamela Flores. Monoliths to microservices - migration problems and challenges: A sms. In *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, pages 135–142, 2021.

[16] Daniele Wolfart, Wesley K. G. Assunção, Ivonei F. da Silva, Diogo C. P. Domingos, Ederson Schmeing, Guilherme L. Donin Villaca, and Diogo do N. Paza. Modernizing legacy systems with microservices: A roadmap. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 149–159, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3):36–43, 2018.

[18] Roger Anderson Schmidt and Marcello Thiry. Microservices identification strategies : A review focused on model-driven engineering and domain driven design approaches. In *2020 15th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2020.

[19] Lulai Zhu, Damian Andrew Tamburri, and Giuliano Casale. Radf: Architecture decomposition for function as a service. *Software: Practice and Experience*, 54(4):566–594, 2024.

[20] R. Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–475, 2017.

[21] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157:110380, 2019.

[22] Mario Dudjak and Goran Martinović. An api-first methodology for designing a microservice-based backend as a service platform. *Inf. Technol. Control.*, 49:206–223, 2020.

[23] Maria Korolov. Why securing containers and microservices is a challenge. 2018.

[24] Anusha Bambhore Tukaram, Simon Schneider, Nicolás E. Díaz Ferreyra, Georg Simhandl, Uwe Zdun, and Riccardo Scandariato. Towards a security benchmark for the architectural design of microservice applications. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Ataollah Fatahi Baarzi, George Kesidis, Dan Fleck, and Angelos Stavrou. Microservices made attack-resilient using unsupervised service fissioning. In *Proceedings of the 13th European Workshop on Systems Security*, EuroSec '20, page 31–36, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Šarūnas Grigaliūnas Algimantas Venčkauskas, Donatas Kukta and Rasa Brūzgienė. Enhancing microservices security with token-based access control method. 2023.

[27] Prajwal Kumar, Radhika Agarwal, Rahul Shivaprasad, Dinkar Sitaram, and K.V. Subramaniam. Performance characterization of communication protocols in microservice applications. pages 1–5, 09 2021.

[28] Carolina Luiza Chamas, Daniel Cordeiro, and Marcelo Medeiros Eler. Comparing rest, soap, socket and grpc in computation offloading of mobile applications: An energy cost analysis. *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2017.

[29] Nikita Lazarev, Neil Adit, Shaojie Xiang, Zhiru Zhang, and Christina Delimitrou. Dagger: Towards efficient rpcs in cloud microservices with near-memory reconfigurable nics. *IEEE Computer Architecture Letters*, 19(2):134–138, 2020.

[30] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula rpc-optimized architecture. In *Proceedings*

of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA '20, page 199–212. IEEE Press, 2020.

[31] Chang Yi, Xiuguo Zhang, and Wei Cao. Dynamic weight based load balancing for microservice cluster. In *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, pages 1–7, 2018.

[32] David R Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, 2004.

[33] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144, 2005.

[34] Ruozhou Yu, Vishnu Teja Kilari, Guoliang Xue, and Dejun Yang. Load balancing for interdependent iot microservices. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 298–306. IEEE, 2019.

[35] You Liang and Yuqing Lan. Tclbm: A task chain-based load balancing algorithm for microservices. *Tsinghua Science and Technology*, 26(3):251–258, 2020.

[36] Yuwei Wang. Towards service discovery and autonomic version management in self-healing microservices architecture. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, pages 63–66, 2019.

[37] Mihai Baboi, Adrian Iftene, and Daniela Gîfu. Dynamic microservices to create scalable and fault tolerance architecture. *Procedia Computer Science*, 159:1035–1044, 01 2019.

[38] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, mar 2012.

[39] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 585–597, 2015.

[40] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, Boston, MA, July 2023. USENIX Association.

[41] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Gagan Somashekar and Anshul Gandhi. Towards optimal configuration of microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, page 7–14, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[44] Pan Fan, Jing Liu, Wei Yin, Hui Wang, Xiaohong Chen, and Haiying Sun. 2pc*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. *Journal of Cloud Computing*, 9:1–22, 2020.

[45] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with sla guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 62–77, 2022.

[46] Reena Panda and Lizy Kurian John. Proxy benchmarks for emerging big-data workloads. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–140, 2017.

[47] Gokul Subramanian Ravi, Ramon Bertran, Pradip Bose, and Mikko Lipasti. Micrograd: A centralized framework for workload cloning and stress testing. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 70–72, 2021.

[48] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, feb 2014.

[49] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Archit. News*, 41(3):475–486, jun 2013.

[50] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery.

[51] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, August 2020.

[52] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Danotia, Mahesh Ketkar, and Christina Delimitrou. End-to-end application cloning for distributed cloud microservices with ditto, 2022.