Cloud Native Daily

Scaling Microservices: A Comprehensive Guide

How to tackle challenges in scaling microservices

Chameera Dulanga Follow 8 min read · Jun 27, 2023

242

Microservices architecture has gained tremendous popularity among developers since it allows them to build highly scalable and flexible applications. However, despite its advantages, scaling microservices can pose significant challenges. For example, monitoring and tracking requests through multiple microservices can become extremely difficult when the number of microservices increases.

As a result, various tools like Helios, Istio, and AWS CloudFormation were introduced to simplify the microservices scaling process. So, in this comprehensive guide, I will discuss some common challenges and tools you can use to prevent them.

. . .

1. Service Coordination

Service coordination is one of the key elements in a microservices architecture. Especially when the number of microservices increases, it becomes challenging to handle communications and interactions become microservices. Hence, you need to understand when there can be issues in service coordination and how to tackle them. Service discovery, load balancing and inter-service communication are the 3 main aspects you need to consider.

Service Discovery

When you have a large number of microservices, it is essential to follow a service discovery pattern to help microservices easily discover each other. But, using centralized service registries may not be the ideal solution to handle the growing number of services efficiently since there can be issues like single-point failure, network traffic, etc. Hence, it is advised to use decentralized service discovery mechanisms to ensure your application is highly scalable. Here are some decentralized service discovery mechanisms you can follow:

- Service mesh frameworks (Istio or Linkerd)
- DNS-Based Discovery (Netflix's Eureka, CoreDNS)
- Self-Registration
- Peer-to-Peer (P2P) Discovery

Load Balancing

Instant demand fluctuations can create an uneven distribution of requests, making some microservices overwhelmed while others remain underutilized. Hence, you must utilize an intelligent load-balancing technique to distribute requests across multiple instances of a service evenly. Dynamic load balancers can adapt to changing traffic patterns and adjust the distribution strategy accordingly. Kubernetes, NGINX, or HAProx are popular tools offering built-in load-balancing capabilities.

Inter-Service Communication

Managing communications between services is another factor you must consider when scaling microservices. With a large number of microservices, managing communication protocols, ensuring reliable message delivery, and handling different data formats have become complex tasks.

So, it would be best to use a reliable communication protocol like [HTTP](https://www.cloudflare.com/learning/ddos/glossary/hypertext-transfer-protocol-http/#:~:text=The%20Hypertext%20Transfer%20Protocol%20(HTTP,of%20the%20network%20protocol%20stack.), gRPC, or message queues to handle the complexity of communication between services. These protocols provide a more reliable, fault-tolerant, asynchronous and scalable way to handle inter-service communications.

. . .

2. Data Consistency

Maintaining data consistency is a significant challenge in applications based on microservices. You must ensure data updates are propagated consistently across services and effectively handle distributed transactions.

Using traditional ACID transactions to ensure strong consistency) is not ideal for data consistency in microservices. The two-phase commit protocol used in ACID transactions can cause significant issues like performance bottlenecks and single points of failure while increasing the design complexity. The eventual consistency model is more suitable for microservices than strong consistency. But, it also has several drawbacks since updating all the data replicas asynchronously takes time.

So, here are some of the actions you can take to ensure data consistency when scaling microservices with minimum drawbacks

- Use the Saga pattern to break down long-running distributed transactions into a sequence of smaller, localized steps.
- Use event-driven architecture to achieve eventual consistency.
- Use CQRS (Command Query Responsibility Segregation) pattern to separate the read and write operations.
- Use optimistic concurrency instead of locking mechanisms to prevent performance issues.

. . .

3. Performance Optimization

Performance optimization is another important aspect you need to consider when scaling microservices. Here are some popular techniques you can follow to ensure high performance at any scale.

- **Identify Performance Bottlenecks:** You can use specialized tools like Helios, New Relic, AppDynamics, and Datadog to profile and monitor microservice performance in real-time to identify bottlenecks.
- **Resource Utilization:** This includes optimizing CPU, memory, and network usage to ensure efficient processing and minimize wastage. You can use caching, connection pooling, and optimizing database queries to reduce resource consumption and improve overall performance.
- **Horizontal Scaling:** Sometimes, adding more microservices and balancing the load between them might be the optimal solution rather than vertically scaling a single microservice.
- **Asynchronous and Non-Blocking Operations:** Use asynchronous and non-blocking operations to handle multiple requests concurrently. Asynchronous processing, event-driven architectures, and non-blocking I/O can significantly reduce response times and improve overall performance.

. . .

4. Selecting the Right Infrastructure

Infrastructure directly affects the scalability of any application since it provides all the resources and tools required to run the application. AWS

Lambda, Kubernetes (K8s), Docker Swarm, and Azure Functions are some of the most popular options. But, you need to evaluate your project requirements with the features provided by each service before making a decision.

Here are some of the crucial factors you need to consider:

- **Workload Characteristics:** Each microservice has different resource requirements, traffic volume, and performance expectations. Hence, you need to evaluate the specific workload characteristics of your microservices and choose an infrastructure that can accommodate those requirements.
- **Skills and Expertise:** Your organization should have the experts to handle the infrastructure. For example, choosing Azure while having AWS experts on your team will make it difficult to manage infrastructure.
- **Cost**: Different infrastructures have varying pricing models and cost implications. Before making a decision, you must evaluate the total infrastructure cost, including upfront costs, operational expenses, and long-term scalability costs.
- **Long-Term Scalability**: Although we can't predict an application's scalability requirements, it is essential to consider long-term scalability to avoid unnecessary complications as the application grows. So, choose an infrastructure that supports the growth of your application with features like auto-scaling; support distributed systems and larger deployments.

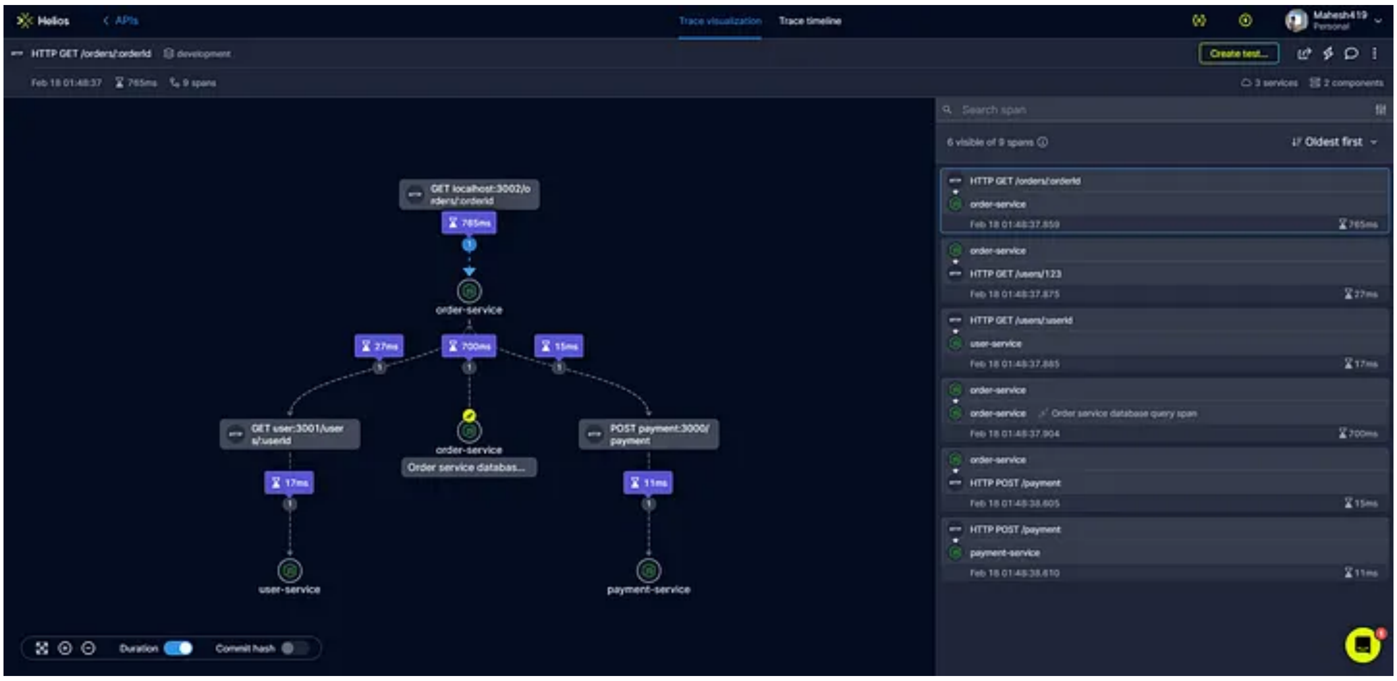
. . .

5. Monitoring and Observability

Monitoring is an important part of any application. It helps developers to gain visibility into their applications and improve the debugging and troubleshooting process. However, due to their distributed nature, implementing a comprehensive monitoring mechanism for microservices can be challenging.

Here are some key aspects to consider, along with examples of monitoring tools that can help:

- **Distributed Tracing:** Distributed tracing provides visibility into the flow of requests and transactions across microservices. With distributed tracing data, you can easily identify performance bottlenecks, latency issues, and dependencies between services. You can use specialized tools like Helios or Datadog to implement distributed tracing for your microservices.



- **Logging and Error Tracking:** Implementing logging mechanisms to capture logs from all microservices allows you to efficiently identify and troubleshoot errors and exceptions.

- Furthermore, you can use error-tracking tools to aggregate and analyze error logs. Helios, Sentry and ELS Stack are popular tools for logging and error tracking.
- **Performance Monitoring**: Monitoring the performance of individual microservices allows developers to identify performance bottlenecks and optimize resource utilization. This can be achieved through tools like Helios, New Relic, and Prometheus that provide detailed metrics and performance profiling.
- **Visualization and Alerting:** Choose monitoring tools that offer visualizations and alerting capabilities. These features help quickly identify performance trends, anomalies, and critical events. Almost all modern monitoring tools support visualizations and alerting.

. . .

6. Service Resilience

Service resilience is a major consideration in scaling microservices to maintain the stability and availability of the entire system. However, as the number of microservices increases, ensuring resilience becomes more challenging due to the distributed nature of the architecture. Here are some key aspects you need to consider to maintain high service resilience:

- **Fault Tolerance:** The distributed nature of microservices makes them vulnerable to various issues like network and hardware failures. You can minimize these issues and improve fault tolerance through redundancy, load balancing, and failover strategies.
- **Failure Handling and Recovery:** When a microservice fails, it is essential to handle the failure gracefully and recover as quickly as possible. For that, you must detect failures promptly, isolate the affected services, and initiate the appropriate recovery actions.
- **Retry Mechanisms:** Retry mechanisms help to handle failures caused by network issues by automatically retrying failed requests. However, it is essential to have an optimal retry configuration (retries count, backoff intervals, etc.) to balance system responsiveness and avoid excessive retries.
- **Resilience Testing:** Make sure to regularly test the resilience of microservices to identify weaknesses and validate the effectiveness of resilience mechanisms. You can use chaos engineering tools or frameworks to systematically test and improve the resilience of microservices.

. . .

7. Deployment Complexity

Deployment complexity is another important aspect you need to consider when scaling microservices. Obviously, things will get more complex since you have to coordinate deployments between microservices and handle configurations for each deployment while minimizing downtime. Here are some steps you can take to ease up the complexity of large-scale microservice deployments:

- **Infrastructure as Code (IaC):** Use IaC tools to manage deployment configurations in a consistent and automated manner. Terraform and AWS CloudFormation are 2 of the most popular IaC tools used with microservices.
- **Continuous Integration and Deployment (CI/CD):** Implement CI/CD pipelines to automate the deployment and release processes. You can use tools like Jenkins, GitLab CI/CD, and CircleCI to automatically build, test, and deploy microservices.
- **Containerization and Orchestration**: Containerization platforms like Docker provide a standardized and portable packaging format for microservices. You can use Kubernetes or Docker Swarm to help manage containerized microservices' deployment, scaling, and lifecycle.
- **Blue-Green or Canary Deployments:** Utilizing deployment strategies like blue-green or canary deployments can help to minimize downtime and reduce the impact of updates. These strategies involve deploying new versions of microservices alongside existing versions, allowing for gradual rollout and testing before switching traffic to the updated version.

. . .

8. Security and Access Control

As the number of microservices increases, the application becomes more vulnerable to security breaches since the attack surface expands. Hence, it is essential to implement robust security measures simultaneously as you scale the microservices. To enhance security in scaling microservices, organizations can take several steps.

- Implementing strong authentication and authorization mechanisms. OAuth 2.0, JSON Web Tokens and Role-Based Access Control (RBAC) are common authentication and authorization solutions you can use.
- Securing inter-service communication through encryption and authentication protocols. You can use techniques like Transport Layer Security (TLS), Mutual TLS (mTLS) or API Gateways to increase inter-service security.
- Regularly patching and updating software. You can use automated patch management tools like Microsoft WSUS, Red Hat Satellite or Ivanti Patch to simplify the patch management process.
- Conducting security audits and penetration testing with tools like Metasploit, OpenVAS and Nessus.

. . .

Conclusion

High scalability is one of the critical features of microservice architecture. But, it is not a silver bullet, and you must consider many areas before deciding.

This article discussed 8 different areas developers need to consider when scaling microservices, including challenges and solutions to avoid them. Apart from what we discussed, there can be various other areas you must consider based on the project requirements. For example, limitations due to tech stack and organizational complexity are 2 other common areas

I hope these suggestions will help you to scale your microservices architecture successfully. Thank you for reading.

. . .

Further Reading:

5 Microservices Challenges and Blindspots for Developers

Microservices dev top 5 challenges and solutions: testing, observability, troubleshooting & debugging, services...

gethelios.dev

Microservices Monitoring: Cutting Engineering Costs and Saving Time

A few ways fort leveraging Helios to save on engineering costs and dev time for a more resource-efficient organization...

gethelios.dev

Testing Microservices - Trace Based Integration Testing Example

Microservices architectures require a new type of testing. Here's why traditional testing fail and the new automated...

gethelios.dev

Top 11 Tools for Microservices Backend Development in 2023

python.plainenglish.io

7 Best Tracing Tools for Microservices

Decide The Best Tracing Tools For Your Microservices Architecture

medium.com

OpenTelemetry: A full guide

Learn all about OpenTelemetry OpenSource and how it transforms microservices observability and troubleshooting

gethelios.dev

OpenTelemetry Tracing: Everything you need to know

OpenTelemetry tracing is filling the gaps of traditional observability methods in microservices apps. Here's how it's...

gethelios.dev

Serverless observability, monitoring, and debugging explained

Serverless troubleshooting requires E2E observability, through collecting trace data on top of logs and metrics- Here's...

gethelios.dev

Microservices AWS Kubernetes Load Balancing Jenkins

Published in Cloud Native Daily

Follow

1.3K followers · Last published Apr 4, 2025

A blog for Devs and DevOps covering tips, tools, and developer stories about all things cloud-native

Written by Chameera Dulanga

Follow

5.8K followers · 177 following

Associate Technical Lead | AWS Community Builder (x4) | Content Manager

No responses yet

Ishu

What are your thoughts?

More from Chameera Dulanga and Cloud Native Daily

In Bits and Pieces by Chameera Dulanga

3 Ways To Detect Circular Dependencies In JavaScript…

How to Prevent Circular Dependencies?

Jan 29, 2024 390 4

In Cloud Native Daily by Maryam Naveed

Blue-Green Deployments with Kubernetes: A Comprehensive…

In the modern world of software development, deploying new versions of…

May 7, 2023 111 10

In Cloud Native Daily by SANKET RAI

Monitoring Kubernetes Pods Resource Usage with…

In this article, we will explore how Prometheus and Grafana can be leverage…

May 20, 2023 53 1

In Bits and Pieces by Chameera Dulanga

Implementing Saga Pattern in a Microservices with Node.js

How to Implement Saga Pattern with Node.js

Jan 3, 2024 1K 4

See all from Chameera Dulanga

See all from Cloud Native Daily

Recommended from Medium

AlgoWing

What Senior Architects Know About SAGA That Juniors Don't

When I was a junior engineer, I thought the SAGA pattern was magic. Every blog and...

Oct 21 145 3

In Stackademic by Shanvika Devi

Stop Saying 'Immutable Means Can't Change'—Java Interviewe...

The one concept that separates "I know Java" from "I understand Java."

4d ago 302 8

In ITNEXT by Animesh Gaitonde

Scaling Distributed Counters: Designing a View Count System...

Architecture, challenges and bottlenecks in counting views at scale

Jul 19 650 13

Abhinav

A Senior Engineer's Guide to Kafka Design Patterns

Apache Kafka has become the backbone of modern event-driven architectures. But...

Jul 15 66

Gaddam.Naveen

5 Threads That Saved My Microservice from a 2 AM Outa...

if you are not a medium member then Click here to read free

Oct 21 11

The Latency Gambler

Design Patterns in Golang That Will Make You a 10x Backend...

Design patterns are tried-and-tested solutions to common software design...

Apr 29 222 7

See more recommendations