

FreeWheel Biz-UI Team

Cloud-Native Application Architecture

Microservice Development Best Practice



中国工信出版集团



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



Springer

Cloud-Native Application Architecture

FreeWheel Biz-UI Team

Cloud-Native Application Architecture

Microservice Development Best Practice



Springer



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONIC'S INDUSTRY
<http://www.phei.com.cn>

FreeWheel Biz-UI Team
Biz-UI
FreeWheel
Beijing, Beijing, China

ISBN 978-981-19-9781-5 ISBN 978-981-19-9782-2 (eBook)
<https://doi.org/10.1007/978-981-19-9782-2>

Jointly published with Publishing House of Electronics Industry, Beijing, PR China
The print edition is not for sale in China (Mainland). Customers from China (Mainland) please order the
print book from: Publishing House of Electronics Industry.
ISBN of the Co-Publisher's edition: 978-7-121-42274-4

© Publishing House of Electronics Industry 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publishers, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publishers nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publishers remain neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721,
Singapore

Foreword

As a veteran in the technology field, I have witnessed the dramatic evolution of the software industry in terms of architecture in the past 20 years. I clearly remember when I first joined IBM China Software R&D Center in 2004, the whole team was focused on building projects based on SOA (Service Oriented Architecture). At that time, SOA was still very heavy, and the implementation relied strongly on J2EE, Web Service, SOAP, and other technologies, which had high learning costs for developers and complex implementation and adoptions; customers also needed to take more time to learn how to use the system. In addition, the technology stack was relatively lacking in support for non-functional requirements (such as resilience, performance, and scalability) that needed to be considered in the design phase and eventually implemented, which was a heavy task. Nevertheless, this brought a lot of inspiration and thinking in the system architecture design of the software industry. As a part of the architectural evolution process, SOA has been an important catalyst for the emergence of new technologies and architectural design styles dedicated to decoupling the complex software systems.

Nowadays, microservices and cloud-native technologies have become the mainstream of the software development industry. Many companies, organizations, and teams are migrating or planning to adopt their business to the cloud and refactor their system architecture based on microservices. In fact, regardless of SOA or microservices, the goal is to decouple the complex system and define and wrap the functional modules in a “service,” to realize the independence of system design, development, and maintenance.

But there are two sides to everything. While we benefit from new technologies, we also need to face lots of challenges introduced by them, such as distributed transactions, networking issues, and service discovery. These are only a small part of the problems faced in building microservices architecture. The main process of software design is trade-off under various constraints.

As the leader of global online video advertising technology and innovation, FreeWheel has strong experience of building microservice application. Since 2016, the team has been committed to refactoring the architecture, and after

4 years hard work, the core business system has been rebuilt based on the microservices and migrated to the cloud. In the process of upgrading the technical architecture, we inevitably encountered many unexpected technical challenges, and the team keeps exploring and moving forward all the way, accumulating a lot of valuable experience. This book is derived from our best practice; the colleagues who participated in writing the book are all responsible for development in the frontline and have in-depth understanding of technical details.

Therefore, I highly recommend that the developers interested in microservices and cloud-native technologies read this book, as it will certainly provide a valuable reference for your work.

FreeWheel, Beijing, China

Qiang Wang

Preface

Practice makes real knowledge. As engineers in the front line, the author team knows the value of real cases to readers and the importance of practice for technical learning. Therefore, most of the cases described in the book come from user scenarios. This is why the book is named “Best Practices.” Of course, we should not underestimate the importance of theoretical learning, and the author team tried to explain accurately the technical concepts based on lots of research, so that the practice can be justified and followed.

The content of the book covers full development lifecycle. No matter what architecture style is used to build a system, it is bound to go through the complete process from design to deployment, and the same is true for microservice applications. Especially with the addition of cloud-native technologies, the development approach and design mindset will be different in many aspects such as technology selection, implementation, and deployment. Therefore, we do not talk about these technologies in a discrete way, but introduce the knowledges of the software development lifecycle step by step based on the development process, in order to bring readers a reasonable and smooth reading experience. From technology selection to service splitting; from agile development to code management; from service governance to quality assurance, the corresponding technologies and practices are shown to readers how to integrate cloud-native technologies into each part of the software development lifecycle.

Sharing customized case. Another feature of this book is that it summarizes some customized practices and tools based on the team’s own experience, such as a low-code development platform for building serverless, a virtual team of middle platform, and an interesting bug bush activity. We believe these will give readers a new feeling. We also hope that these special practices will help you improve your own R&D tools.

This book can be used as a professional book or reference book for industrial practitioners who wish to use the concept of cloud-native technology in practical applications; it should also be useful to senior undergraduate and postgraduate students who intend to access the industry, researchers who work on cloud-native

application architecture and microservices and may also gain inspiration from this book.

Beijing, China
Nov 2022

FreeWheel Biz-UI Team

Introduction to the Author Team

About FreeWheel Biz-UI Team

FreeWheel Biz-UI team started to build cloud-native application based on microservice architecture from 2017 and has accumulated rich experience. In addition to developing enterprise-level SaaS, they also lead internal innovative projects such as microservice business middle platform, service governance platform, and serverless low-code development platform.

The team is committed to the popularization and promotion of cloud-native technologies and is keen to share their years of engineering practice. They have established a technical column on microservices practice in InfoQ and published many technical articles. Members of the team have published several technical patents based on their development practices, which have passed the audit of the USPTO. The team also has many experts in cloud-native domain, published books and shared their practices in QCon, AWS Summit, CNCF Webinar, and other offline and online technical conferences.

The authors of this book all come from the team, and they are all experts in the relevant technologies of the chapters they are responsible for, with many years of development experience and a love for technology sharing. The book on microservices best practices covers several cloud-native technology hotspots and provides a detailed summary of the whole development lifecycle practices, which can provide readers with valuable reference.

About FreeWheel

FreeWheel, A Comcast Company is a provider of comprehensive ad platforms for publishers, advertisers, and media buyers. Powered by premium video content, robust data, and advanced technology, the company is revolutionizing the way publishers and marketers transact across all screens, data types, and sales channels.

List of Authors

Ruofei Ma, Lucas Wang, James Wang, Yu Cao, Kan Xu, Fan Yang, Qi Zhang,
Yuming Song, Yanmei Guo, Hanyu Xue and Na Yang

Acknowledgements

We would like to thank Rufei Ma, the software architect of FreeWheel's BizUI team, as the first author of this book, he led the team to write and polish the chapters and contents of the book. Thanks to the author team for their contributions, they are: Ruofei Ma, Lucas Wang, James Wang, Yu Cao, Kan Xu, Fan Yang, Qi Zhang, Yuming Song, Yanmei Guo, Hanyu Xue and Na Yang. We would also like to thank all the colleagues who participated in building microservices and cloud adoption for their hard work. We also want to thank our families, whose support made it possible for us to finish the book. In addition, we would like to thank the editor of Springer, whose professionalism is the key to improve the quality of the book.

This book was born in the beginning of the New Year, which signifies the renewal of everything for a year. We also hope this book can help more readers to build a scalable and maintainable cloud-native application from scratch in a brand new way, and wish everyone can easily build enterprise applications with microservices technology, and really enjoy the convenience brought by microservices and cloud-native technology efficiently!

About the Book

This book introduces the engineering practice of building microservice applications based on cloud-native technology, and there are nine chapters in the book, each chapter is briefly described as follows.

Chapter 1: Microservices in the Cloud-Native Era

This chapter starts from the characteristics of microservices and provides an in-depth analysis of the concepts and core technologies of cloud-native, and what changes microservice applications need to make in the cloud-native era, to complete the development journey from traditional microservices to cloud-native applications.

Chapter 2: Microservice Application Design

In this chapter, we will discuss how to design a microservice application based on the team's practice. The author will talk about the architectural selection of the application and introduce the solutions of architecture, presentation layer, persistence layer, and business layer and will also analyze how to adopt microservice from monolithic systems.

Chapter 3: Service Development and Operation

This chapter will introduce how to go through the whole development process by Scrum agile development method based on the team's engineering practice and introduce the service management and operation and maintenance platform we built to improve the development efficiency.

Chapter 4: Microservice Traffic Management

Service mesh is the preferred solution for traffic management in the cloud-native era. With declarative configuration, you can give your application the ability to control traffic with transparency. This chapter details how to use it to provide traffic control for your microservice applications, based on our practice with service mesh.

Chapter 5: Distributed Transactions

As software systems move from monolithic applications to microservices and cloud-native, and the trend of database decentralization, local transactions on monolithic

applications will transform into distributed transactions, posing a challenge to the need for data consistency. This chapter will introduce our team's practice of using Saga pattern to implement distributed transactions.

Chapter 6: Serverless Architecture

The advantages of building elastic and scalable applications through serverless computing are becoming more and more obvious. As an emerging application architecture, what are its core concepts, what are its features that distinguish it from traditional architectures, its advantages and application scenarios, and what changes it can bring to the building of applications? This chapter will explain each of these issues.

Chapter 7: Service Observability

There are many challenges when using microservice architecture and migrating to the cloud, especially how to know the status and behavior of the application, how to quickly find and solve online problems, and how to monitor the invocation chain between services, all of which will have to be faced. Building observable applications is an important factor in ensuring service quality. This chapter will introduce the definition and application of service observability.

Chapter 8: Quality Assurance Practices

In this chapter, we will introduce some of the practical experiences related to quality assurance accumulated in the process of building microservice applications, and tell how the team can build a quality assurance system in the cloud-native era through sound testing and chaos engineering.

Chapter 9: Continuous Integration and Continuous Deployment

Continuous integration and continuous deployment are necessary to build cloud-native applications. In this chapter, we will talk about the automation triggering, differential execution, and unified product archiving of continuous integration and introduce the product release planning and cloud-native-based deployment framework after microservicization, as well as the full lifecycle support of continuous deployment for microservice applications.

Contents

1	Microservices in the Cloud Native Era	1
1.1	Starting with Microservices	1
1.1.1	Characteristics of a Microservice Architecture	2
1.1.2	Microservice Trade-Offs	7
1.2	Cloud-Native Applications	10
1.2.1	What Is Cloud-Native	11
1.2.2	Cloud-Native Technologies	14
1.2.3	Characteristics of Cloud-Native Applications	18
1.3	From Microservices to Cloud-Native	20
1.3.1	Adjustment of Nonfunctional Requirements	20
1.3.2	Change in Governance	21
1.3.3	Deployment and Release Changes	22
1.3.4	From Microservice Applications to Cloud-Native Applications	23
1.4	Summary	25
2	Microservice Application Design	27
2.1	Application Architecture Design	27
2.1.1	Service Architecture Selection	27
2.1.2	Service Communication Strategy	34
2.1.3	Storage Layer Design and Selection	42
2.2	Legacy System Modification	45
2.2.1	Greenfield and Brownfield	46
2.2.2	Strangler Pattern	47
2.3	Business Logic Design	52
2.3.1	Splitting Services	53
2.3.2	Design API	60
2.4	Summary	65

3 Service Development and Operation	67
3.1 Agile Software Development	67
3.1.1 From Waterfall Model to Agile Development	68
3.1.2 Scrum in Practice	70
3.2 Runtime Environment	76
3.2.1 Development Environment	76
3.2.2 Test Environment	78
3.2.3 Staging Environment	79
3.2.4 Production Environment	79
3.3 Code Management	80
3.3.1 Git Branch Management	80
3.3.2 Code Inspection Based on Sonar	84
3.3.3 Code Review	87
3.3.4 Commit and Merge	89
3.4 Low-Code Development Platform	90
3.4.1 Low Code and Development Platform	91
3.4.2 Low-Code Development Platform Practices	91
3.5 Service Operation and Maintenance Platform	96
3.5.1 Problems to Be Solved by the Platform	96
3.5.2 Platform Architecture	97
3.5.3 Platform Modules	97
3.6 Service Middle Platform	101
3.6.1 What Is a Middle Platform	101
3.6.2 The Road to Building a Middle Platform	103
3.7 Summary	108
4 Microservice Traffic Management	109
4.1 Traffic Management in the Cloud-Native Era	109
4.1.1 Flow Type	110
4.1.2 Service Mesh	111
4.2 Service Discovery	113
4.2.1 Traditional Services Discovery Problems on the Cloud	114
4.2.2 Kubernetes' Service Discovery Mechanism	115
4.3 Using the Istio Service Mesh for Traffic Management	117
4.3.1 Core CRDs	118
4.3.2 Istio-Based Traffic Management Practices	128
4.3.3 Common Adoption Problems and Debugging	136
4.4 Improving Application Fault Tolerance with Istio	145
4.4.1 Circuit Breaking	145
4.4.2 Timeouts and Retries	149
4.5 Summary	152

5	Distributed Transactions	153
5.1	Theoretical Foundations	153
5.1.1	Background	153
5.1.2	ACID: Transaction Constraints in the Traditional Sense	156
5.1.3	CAP: The Challenge of Distributed Systems	157
5.1.4	BASE: The Cost of High Availability	158
5.1.5	Write Order	158
5.2	Solution Selection for Distributed Transaction Framework	159
5.2.1	Existing Research and Practice	159
5.2.2	Design Goals of Distributed Transaction Framework	164
5.2.3	Choosing Saga	165
5.2.4	Introducing Kafka	166
5.2.5	System Architecture	169
5.2.6	Business Process	170
5.3	Distributed Transactions Based on Saga and Kafka in Practice	171
5.3.1	Improvements to Kafka’s Parallel Consumption Model	171
5.3.2	Deployment Details	173
5.3.3	System Availability Analysis	173
5.3.4	Production Issues and Handling	174
5.4	Chapter Summary	177
6	Serverless Architecture	179
6.1	What Is Serverless Architecture	179
6.1.1	Definition of Serverless Architecture	179
6.1.2	Development of Serverless Architecture	181
6.1.3	Advantages of Serverless Architecture	182
6.1.4	Shortcomings of Serverless Architecture	183
6.2	Serverless Architecture Applications	185
6.2.1	Building Web API Backend Services	185
6.2.2	Building the Data Orchestrator	187
6.2.3	Building Timed Tasks	188
6.2.4	Building Real-Time Stream Processing Services	189
6.3	Serverless Architecture in Practice	192
6.3.1	Why AWS Lambda	192
6.3.2	Import and Processing of Large Amounts of Data	193
6.3.3	Log Collection and Processing	201
6.4	Summary of This Chapter	211
7	Service Observability	213
7.1	What Is Observability	213
7.1.1	Definition of Observability	213
7.1.2	Three Pillars of Observability	214
7.1.3	The Difference and Relation Between Observability and Monitoring	215
7.1.4	Community Products and Technology Selection	217

7.2	Logging Solutions Under Cloud-Native	218
7.2.1	Log Classification and Design	218
7.2.2	Evolution of Cloud-Native Log Collection Scheme	227
7.2.3	Displaying Logs with Kibana	234
7.3	Distributed Tracing	246
7.3.1	Core Concepts of Distributed Tracing System	246
7.3.2	Jaeger-Based Tracing Solution	247
7.4	Metrics	255
7.4.1	Collecting Metrics with Prometheus	256
7.4.2	Displaying Metrics with Grafana	264
7.5	Monitoring and Alerting	266
7.5.1	Monitoring Platform	266
7.5.2	Alert System	277
7.6	Summary	281
8	Quality Assurance Practices	283
8.1	Quality Assurance System	283
8.1.1	Quality Challenges	284
8.1.2	Testing Strategy	285
8.1.3	Building a Quality Assurance System	287
8.2	Testing Practices	290
8.2.1	Unit Testing and Mock Practice	290
8.2.2	Godog-Based Integration Testing Practices	296
8.2.3	Cypress-Based End-to-End Testing Practices	302
8.2.4	Test Automation	305
8.3	Chaos Engineering	309
8.3.1	The Core Concept of Chaos Engineering	309
8.3.2	How to Run Chaos Experiments	316
8.3.3	Fault Injection Experiments with System Resources	323
8.3.4	Service Mesh-Based Network Traffic Fault Injection Method	333
8.4	Quality Assurance of Production-Like Environments	338
8.4.1	Monitoring and Analysis of Online Services	339
8.4.2	Bug Bash Practice	341
8.4.3	Post-release Check Practice	344
8.4.4	Disaster Recovery Strategies and Practices	346
8.5	Summary	350
9	Continuous Integration and Continuous Deployment	351
9.1	Git-Based Continuous Integration	351
9.1.1	Auto-Trigger Pipeline	352
9.1.2	Pipeline Differentiation and Unified Collaboration	359
9.1.3	Pipeline Product Storage Planning	363
9.2	Helm-Based Continuous Deployment	365
9.2.1	Deployment Planning	366

9.2.2	Framework for the Multiple Clusters Deployment in Different Environments	368
9.2.3	Cloud-Native Support and Task Maintenance	374
9.3	Continuous Deployment Practices Based on Kubernetes	377
9.3.1	Pod Resource Quotas and Horizontal Autoscaling	377
9.3.2	Service Online/Offline Workflow and Fault Analysis	379
9.4	Summary	382

Chapter 1

Microservices in the Cloud Native Era



Nowadays, microservice architecture has almost become the first choice for all companies and teams to build applications. After years of evolution, it is becoming more and more mature. Even teams are not yet using microservices for now, they are planning to adopt microservices either. However, with the rapid popularity of cloud-native technologies and ecosystems in recent years, traditional microservices applications are changing. How to build a microservices application based on cloud-native has become the primary technical challenge for developers.

In this chapter, we will talk about the characteristics of microservices, introduce the concepts and core technologies of cloud-native, deep dive what changes microservices applications need to do in the cloud-native era to face the challenges of this technological trend, and then summarize the development process of changing from traditional microservices to cloud-native applications.

1.1 Starting with Microservices

In 2012, the term, microservices first appeared in public. If we regard that as the first year of microservices, it has been a journey of over 10 years. Everyone has a different understanding of microservice architecture, and we tend to agree with a more authoritative definition from software engineering master, Martin Fowler:

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

Microservice architecture provides a new mindset for software development, which is more suitable for the business scenario of frequent requirement changes and rapid product iteration. The independent services bring more flexibility to each

development phase, such as building, testing, and deploying, and also reduce the system resource cost for applications through a more fine-grained scaling mechanism. Of course, the “silver bullet” of software development has not yet appeared, and microservice architecture is also not a savior, it has pros and cons. In this section, let’s go through what its key features are.

1.1.1 Characteristics of a Microservice Architecture

In his article, Martin Fowler lists a series of features of microservice architecture, and I think the following points are the most representative.

Componentization via Service

Decoupling has always been the goal pursued in building software. Even if a monolithic application is logically a whole structure, it is usually split into different modules depending on its functionality (assume it is built by a good architect). These relatively independent functional modules can be understood as components. Unlike monolithic applications, services in a microservice application become components and are out-of-process components that interact with each other through a network.

One of the advantages of services as components is that it makes deployment relatively independent. In-process applications such as a monolithic application need to be packaged and deployed as a whole, even if only a small part of it changed, whereas a microservice application does not have to. Of course, not all services of the application must be completely independent; the more complex the business logic, the more dependencies.

Independent services allow for clearer invocation between services, as services must interact with each other through a commonly agreed contract, such as API. The way in-process modules are invoked does not have this restriction, and if the visibility of functions is designed improperly, they are likely to be used arbitrarily by developers with bad habits, and eventually it is running far away from the goal of loose coupling that was intended.

In addition, with the change from in-process calls to out-of-process calls, the strong stickiness between components is also destroyed by the network, and a series of nonfunctional requirements are introduced; for example, the original local transactions become distributed transactions, or communication timeouts and retries have to be implemented to ensure the reliability of the system.

Organized Around Business Capacities

In his book, Microservice Architecture Design Patterns, the master of software development Chris Richardson writes, “The essence of microservices is the splitting

and definition of services, not the technology and tools.” The rationality of service splitting determines the rationality of interaction between services, which in turn affects the efficiency of application development. In my opinion, building an application based on business is the most difficult and the most important step. The functionality of the application to be developed comes from the client’s requirement, which is called business, and the reason why business is split into different parts is that they are loose coupling with each other. Therefore, it becomes a natural choice to build the application around the business, and decoupling is easy to achieve. This is why domain-driven design has come back into vogue after microservices became popular.

However, because of Conway’s Law, application development is largely limited by the structure of the organization. We can simply interpret Conway’s Law as “the organization determines the architecture.” As shown in Fig. 1.1, a team will build a system that is consistent with the organizational structure. An organization divided by different functionalities (e.g., front-end, back-end, DBA, Ops) will most likely develop a multiple-layer system. Trying to build modules based on business will encounter some difficulty to cross (e.g., communication, integration), and reorganizing the organization is usually unlikely, which is why service (business) splitting is one of the biggest difficulties in microservice development.

Our team used a relatively tricky approach when facing this problem. At the beginning of the development, we made a rearrangement of the functional division, where regular positions like front-end, back-end, and QA were originally eliminated and engineers needed to develop in a full-stack role. The support teams, such as DBA and Ops, were also assigned to virtual teams according to different lines of business in order to complement the development team.

It is a very valuable rule to build systems around the business. When you are confused in the process of splitting services, think carefully about this principle and you may have the answer.

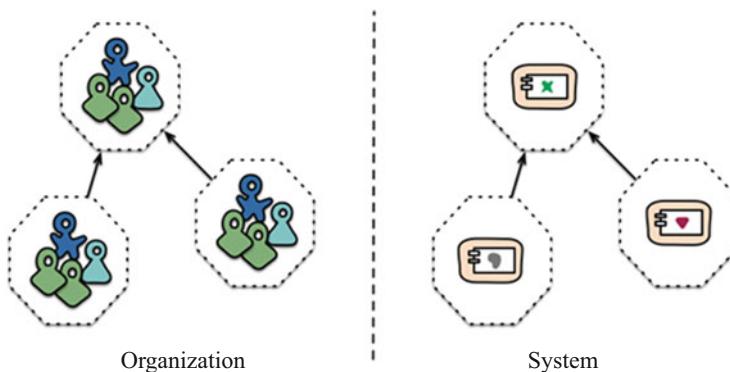


Fig. 1.1 Service boundaries reinforced by team boundaries (source from <https://martinfowler.com/articles/microservices.html>)

Decentralized Governance and Data Management

Another feature that comes with splitting services is decentralization; we will introduce decentralized governance and data management as below.

Decentralized Governance

Each business requirement contained in an application must be different, and different business models must have the most suitable technical solution for it. By splitting, we can have different choices in technical solutions when implementing different services, which is called heterogeneity. Using a suitable solution is more important than using a unified solution because it is more efficient, for example, using object-oriented languages to build services with complex business models for easy modeling, and using Golang language to build middleware services for performance and efficiency. The same is true at the storage layer. For business models with cascading relationships, it is more appropriate to use a documented storage solution like MongoDB, and for big data offline analytics, it is more appropriate to use a data warehouse solution. For development teams, this decentralized build approach is also more flexible.

In contrast, centralized governance will produce a unified technology standard, but this standard is not suitable for all business teams, who may need to make compatibility and adjustments to address issues where the technology solution does not meet the requirements, than it brings extra effort. Of course, unifying the technical stack will reduce the dependency on technology and maintenance costs, but in the context of cloud-native, we've noticed that the way of simplifying the technical stack is gradually becoming less and less, because the complexity and nonfunctional requirements of applications are getting more and more sophisticated, and the technique ecosystem is improving day by day, and the development teams do not have extra burden by involving new technologies or tools. Therefore, for microservices, this divide-and-conquer decentralized governance becomes one of the important features.

Data Management

Data is the embodiment of the business model in the storage level; in other words, essentially data is the business. Therefore, data management also reflects the same idea of building services around the business. This is often represented by different services using different databases, different tables, or using different storage solutions. Figure 1.2 shows the different forms between microservice architecture and monolithic applications at the storage layer.

Monolithic applications typically (but not absolutely) use a logical single database to store data. If you're a senior programmer with many years of development

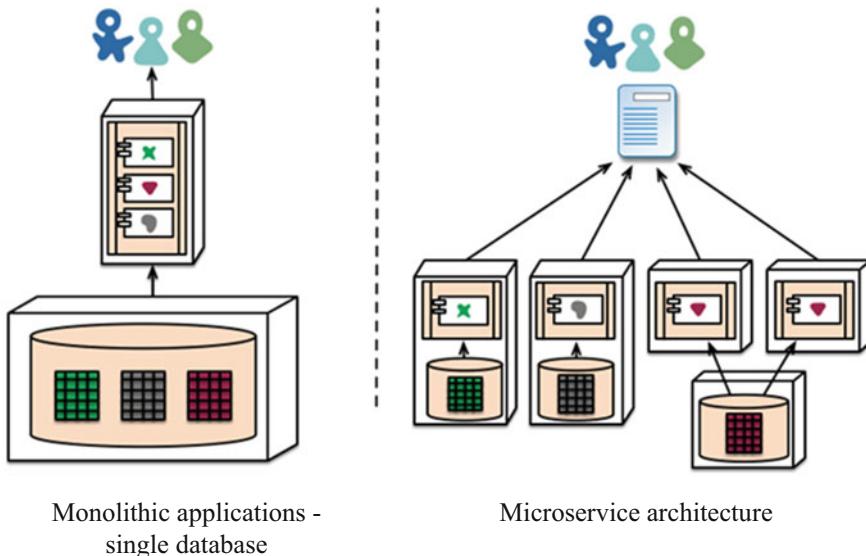


Fig. 1.2 Decentralized Data Management in microservices (source from <https://martinfowler.com/articles/microservices.html>)

experience, this long-term habit can make you uncomfortable when migrating to the microservice architecture. For example, you'll have to be careful to split the database tables corresponding to your business module to ensure they are arranged correctly.

The benefits of decentralized data management remain flexibility and coupling, where business changes at the data level can be handled by the service itself. But the disadvantage is also very obvious: it brings the problem of distributed transactions. Business scenarios are connected to each other, and it is difficult to have a complete chain of calls without transactional operations. Because of the CAP theory, there is no perfect solution for the distributed transaction, and developers have to make trade-offs based on application scenarios. As a result, microservice architecture tends to be designed to enable nontransaction collaboration between service-to-service or to solve the problem with eventual consistency and compensation mechanisms. Chapter 5 of this book will introduce our best practice on distributed transactions, which can be used as a reference for readers.

Infrastructure Automation

Essentially, the pipeline is the core of automated Continuous Integration (CI) and Continuous Delivery (CD). Pipelines allow us to move code step by step from the development environment to the production environment. As cloud platforms become more and more capable, the complexity of building, deploying, and maintaining microservices is gradually decreasing. The author believes that

automated continuous integration and continuous delivery are not necessary for building microservices, but they are very important for efficiency. Imagine if an application includes hundreds of services, without an automated continuous integration and continuous delivery pipeline, it would be a huge hassle to deploy these services effectively. On the other hand, one of the main purposes of splitting an application into services is to deploy them independently to improve operation efficiency and system resource utilization. If the lack of automated deployment leads to inefficient operations and maintenance, this defeats the original design intent. Therefore, building an automated continuous integration and continuous delivery pipeline for microservice architecture, especially in the cloud-native era, becomes a very important point.

Design for Failure and Evolutionary Design

For a microservice architecture, there are two things to keep in mind at the design level.

Design for Failure

As mentioned earlier, after splitting the business module into services, the invocation between services is likely to fail for various reasons, such as network jitter, unavailability of upstream services, traffic overload, routing errors, etc. This requires us to fully consider these issues when we build microservice applications and then find a way to minimize the impact on users when failures occur. Obviously, this will increase the development burden and we need to implement more nonfunctional requirements, or control logic, and this should be the biggest disadvantage of a distributed architecture like microservices over a monolithic application.

To solve the problem, service governance and easier management of communication between services through service mesh becomes an important topic. When failures happen, this technique can detect and find failures in time and recover them automatically. With the addition of cloud-native technologies, it becomes easier to implement failure-oriented design. For example, we can complete automated restart based on probes and operation policies or build various service monitors through logs, metrics, and tracing.

Evolutionary Design

We often jokingly say that the only thing that is immutable in software development is change. This statement is a bit self-deprecating but true. Frequent changes in requirements and strategic adjustments to keep up with the market require us to design applications with the ability to evolve. When we split an application into services, it could change independently, and we can make individual adjustments to

changes without having to rebuild the entire application, for example, by continuously and in batches updating different services, gradually making changes continuously for various parts of the application, which is called evolution. We can even develop applications in an abandoned way. For example, if a feature is developed for a certain commercial event, it can be built as a separate service and deleted it when the campaign is over, without affecting the main business part. To make the evolution more smooth and more logical, we also need to separate the stable parts from the unstable parts to ensure that the impact of changes is minimized.

It can be said that microservice provides a prerequisite for designing evolved applications; separate services make evolved design easier to achieve.

1.1.2 Microservice Trade-Offs

Trade-offs are the main behavior of software design and development, and we always have to make trade-offs based on a variety of constraints. There are trade-offs in technique selection, trade-offs in whether a program takes more time or more system resources, trade-offs in performance and functionality, and even trade-offs in naming a function. Similarly, there are also trade-offs in the use of microservice architecture. In this section, we'll list a few of its key advantages and disadvantages, and the corresponding trade-offs, to help you aim your goal in microservices development.

The Border Effect Brought by Services

When you split a business module into services, the sense of boundaries between the parts becomes stronger. Because cross-service calls are more costly than in-process function calls, you need to think about communication, latency, and how to handle failed calls. And it's easy to get around such boundary constraints in monolithic applications, and if the development language used happens to have no visibility constraints, disordered calls can break the modular structure and couple between them. Another source of such coupling is the centralized database, where different modules can access data at will, which is usually the main source of coupling in complex business systems.

The decentralized data partition of microservices avoids this scenario. Accessing data through APIs exposed by services makes it easier for code to implement clear business scope, while avoiding disparate modules in a mess of dependencies.

The boundary effect from servitization is beneficial in that it ensures isolation between services. The prerequisite is to split services, i.e. the boundary definition is reasonable. If the right boundaries are not controlled in the design phase, this advantage can turn into a defect instead. When you find that you always need to access data through cross-service calls, it means that there is probably a problem with the boundary definition. In terms of complexity, even though services are

smaller business units and easier to understand, the overall complexity of the application is not eliminated, and it is simply transferred to service calls. The length of the call chain can make this complexity manifest. So well-defined service boundaries can reduce this problem. Sub-domains and bounded contexts in domain-driven design can help us define boundaries by allowing us to better understand the isolation of the business. Therefore, when enjoying the benefits of boundary effects, it is important to define it well first.

Independent Deployment

Independent deployment keeps changes to the application within a scope that leaves other parts unaffected, and it also makes evolutionary development possible. When feature development is complete, you only need to test and deploy the services you are responsible for, and even if a bug occurs, it only affects a small part of the business; the whole system won't all fall apart because of your problem. Provided, of course, that needs a good design for failure. With the addition of a cloud-native infrastructure like Kubernetes, the ability to automatically roll back doesn't even let users know that you actually did a failed deployment.

Independent deployments will accelerate the speed of updating applications. However, as more services are added, deployment behavior will occur frequently and the speed of application iteration will be proportional to the speed of delivery. Therefore, having the ability to deploy quickly is an important requirement to ensure continuous microservice application iteration. The advantage behind this is that it allows the application to introduce new features faster and respond quickly to changes in the market. Looking back at my software development experience more than a decade ago, a product release required a rather tedious process that was painful and inefficient. Now, however, with the addition of continuous delivery, daily builds have become the norm.

But independent deployments also put more pressure on operations and maintenance as one application becomes hundreds of microservices. Without an automated deployment pipeline, I'm afraid there's no way to complete updates for tons of services. At the same time, there is more work to manage and monitor the services, and the corresponding complexity of operations and tools increases. Therefore, if you are building infrastructure and tools that do not yet have the ability to automate continuous delivery, microservices are not the right choice for you.

Independent Technical Stack

The relative independence of microservices gives you the freedom to choose the technology stack to implement it. Each service can use different development languages, different libraries, and different data storage, and eventually form a heterogeneous system. One of the biggest advantages of an independent technical stack is that it allows you to choose the most appropriate development language and

tools based on business characteristics in order to improve development efficiency. For example, if your service is mainly responsible for big data analytics, you will likely give preference to those mature big data products based on the Java technique stack. This freedom also brings confidence to the development team and gives them a sense of ownership that they are in charge.

Another advantage is version control. If you've ever developed a monolithic application, I'm sure you've probably had the experience of using a dependent library that needs to be upgraded, but this is likely to affect other modules in the system, and you've had to coordinate, communicate, wait, or even end up with nothing to do about it. And as the size of your code grows, the difficulty of dealing with version issues grows exponentially. Microservices solve this problem perfectly. Any upgrade of a library is only responsible for its own service.

Is it necessarily a good thing to have diversity in technology? Of course not. Most teams are generally encouraged to use only a limited technical stack, and involving too many different technologies can overwhelm a team: first by making it more expensive to learn the technology and to maintain and upgrade it, and also by making it more complicated to build a service, especially by making continuous integration and continuous deployment less efficient. Therefore, our recommendation is to use technologies and tools that are adequate, and to do research and experimentation before using new technologies to provide reference data for selection.

Distributed System

Distribution can improve the modularity of the system, but it also has obvious disadvantages, starting with performance problems. Compared to in-process function calls, remote calls are much slower. Even if the system does not have particularly high-performance requirements, if multiple services have to be called on a transaction, they add up to a significant amount of latency. If it happens that the amount of data being transferred is relatively large, it can slow down the response time even further.

The second problem to face is the failure of upstream services. Because of network intervention, remote calls are more likely to fail, especially with a large number of services and communication, and the number of failure points increases. If the system is not designed for failure in mind, or does not have good monitors, it is easy for engineers to be helpless when problems arise, or even for developers responsible for different services to blame each other and pass the buck, because no one feels that it is their own code that has gone wrong when they cannot find the root cause. That's why in the cloud-native era, observability like full-stack tracing is getting more and more attention from everyone.

The third drawback is the introduction of distributed transactions. This is a major headache and there is no perfect solution. It is important for developers to be aware of consistency issues and develop corresponding solutions in their own systems.

For the problems caused by distribution, we need to use some methods to mitigate them. The first is to reduce the likelihood of failure occurring by reducing the number of invocations. Consider using a batch approach to invocation, where multiple data requests are completed during a single invocation. The second approach is to use an asynchronous approach. For services that do not have strong dependencies and do not need to be invoked directly, it makes more sense to drive them through events.

Complexity of Operation and Maintenance

Rapid deployment of a large number of microservices increases the difficulty of operations and maintenance. Without a sound means of continuous deployment and automation, this almost cancels out the benefits of independent deployments. The need for management and monitoring of services also increases, as does the operational complexity. To address this, teams need to fully involve DevOps, including a hands-on approach, tools, and culture.

The author believes that the primary driver of architectural evolution is to reduce costs and improve efficiency; i.e., improving productivity and reducing costs are the primary considerations for architecture selection. In this section, we list several advantages and disadvantages of microservice architecture that need attention and provide corresponding solutions to the problems. One point to emphasize is that the productivity gains from microservices are only applicable to applications of higher complexity; for startup teams or simple applications, monoliths are almost the most efficient choice. The choice of architecture needs to be made after considering various aspects such as team, technical background, and product before coming to a conclusion.

The later parts of this chapter introduce the concepts related to cloud-native applications and provide a reference for you to migrate from traditional microservice applications to cloud-native applications.

1.2 Cloud-Native Applications

More and more enterprises are migrating their applications to the cloud because “onboarding cloud” can reduce resource costs and improve application scalability. Microservice applications have become the first choice for building cloud-native applications. According to authoritative surveys, more than half of respondents said they have used microservice concepts, tools, and methodologies into their development process. Migrating microservices to the cloud has become the primary goal of enterprise IT development in recent years. In this section, we will talk about the concept of cloud-native to give you an accurate understanding of it.

1.2.1 What Is Cloud-Native

The term cloud-native first appeared in public in 2010 when Paul Fremantle, founder and CTO of WSO2, wrote a blog post titled, “Cloud-Native.” The post described the challenges of migrating software to a cloud environment and listed what he considered to be the core characteristics of cloud-native, such as distributed, elastic, multi-tenancy, incremental deployment, and testing. Following this, Matt Stine, former CTO of Pivotal, authored an e-book in 2015 called *Migrating to Cloud-Native Application Architectures*, which has since become widely circulated, which is why the industry generally endorses it. Matt believes that cloud-native application architectures should have the following characteristics:

- Meeting the 12 elements principle
- Using microservice architecture
- Building self-service agile architectures
- API-oriented collaboration
- Anti-vulnerability

As cloud-native technology evolves rapidly, so does the definition of cloud-native. 2019 VMware acquires Pivotal and gives the latest definition of cloud-native:

Cloud native is an approach to building and running applications that exploit the advantages of the cloud computing delivery model. Cloud native development—also increasingly referred to as modern application development—is appropriate for public, hybrid, and private clouds; it's about how applications are created and deployed, not where.

The definition specifically highlights four areas to focus on in cloud-native development, as follows.

- Microservices
- Containerization
- DevOps
- Continuous delivery

When we talk about cloud-native, we can't ignore the Cloud Native Computing Foundation, aka CNCF, a neutral organization founded in 2015 with the initial goal of building a well-established cloud-native ecosystem around Kubernetes. It has grown to become the absolute authority in cloud-native area, with nearly 100 projects incubated in it, and many familiar open source projects have graduated from the CNCF, almost all of which have become de facto standards in related fields, such as Kubernetes and Prometheus. The CNCF defined cloud-native in 2018, and that definition is used to this day.

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

As you can see, based on the definition of CNCF, there are five main technologies of cloud-native concern.

- Containers
- Service mesh
- Microservices
- Immutable infrastructure
- Declarative API

Unlike products, where the definition of a product can be explicitly given by the development team, technology is constantly evolving, especially cloud-native technology, which has changed continuously in 5 years since it started to be proposed in 2015, with emerging concepts such as service mesh and serverless computing quickly becoming important fields of the cloud-native area. As a result, it is difficult to give a precise, unchanging definition of cloud-native technology, and they can usually only be described based on technical characteristics. Everyone can have their own understanding of cloud-native, and its connotation varies from time to time. The author suggests understanding the essence of cloud-native based on these key technical characteristics. One of my views is also given here for your reference.

Using cloud-native technology, we can build scalable and elastic applications on cloud platforms with automated continuous delivery to meet the rapidly changing needs of enterprise businesses.

It is difficult for readers who first see the definition of cloud-native to understand its meaning from the words. Here we help you understand cloud-native by briefly reviewing the development history of cloud computing.

In 2006, Amazon launched its EC2 service, marking the emergence of infrastructure as a service (IaaS). In terms of product form at the time, the cloud platform offered extremely limited capabilities, based only on virtual machine computing resources subsequently also gradually developed into storage, network, and other products of hardware resources.

Around 2010, the concept of platform as a service (PaaS) emerged to enable more software-level services to be provided on top of IaaS, such as operating systems, middleware, etc. As one of the representative vendors, Heroku put forward the famous principle, 12 factors, which also provides reference guidelines for developers to build cloud-based platform applications. At this stage, what needs to be managed by users themselves gradually becomes less and less, and many nonfunctional requirements can be left to the cloud platform, such as load balancing and auto-scaling. PaaS further evolves into software as a service (SaaS), which pursues an almost fully hosted application, and developers only need to focus on the business. Figure 1.3 illustrates the evolution of the cloud platform.

On the other hand, with the emergence of Docker in 2013, container technology became very popular and the container orchestration market became highly competitive, ending with the overall victory of Kubernetes. At the same time, CNCF was

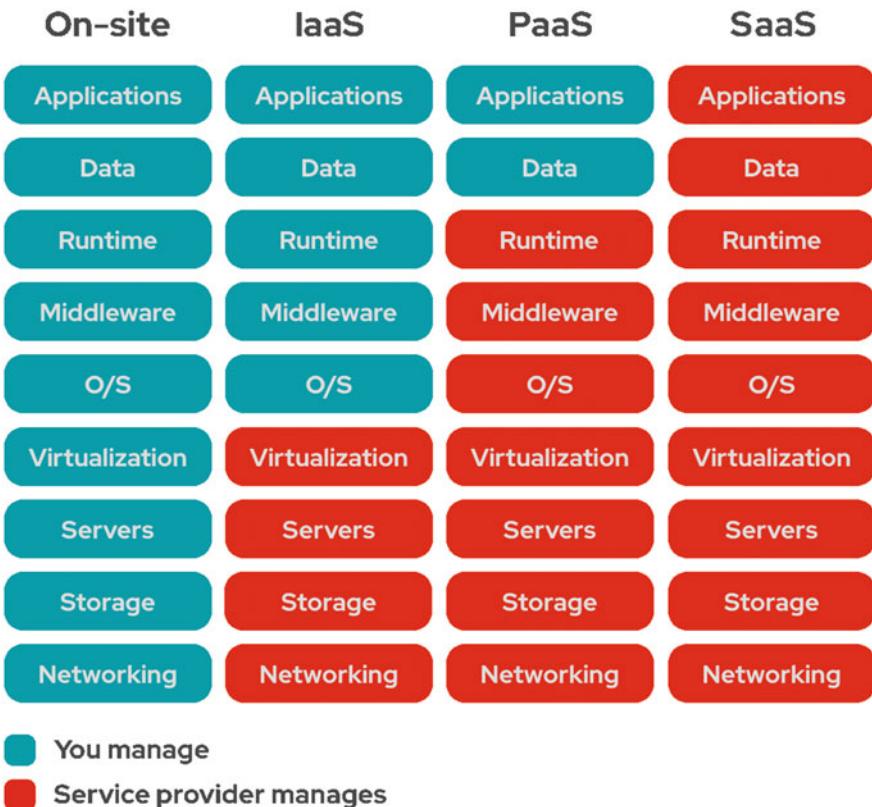


Fig. 1.3 The evolution of the cloud platform (from <https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas>)

founded and started to promote cloud-native technology. As you can see, container technology is very suitable for cloud platforms, and its emergence has changed the way applications are deployed and distributed through PaaS platforms. With containerization and container orchestration, the hosting capabilities of PaaS, cloud-native becomes gradually complete.

The development of cloud-native technology and ecosystem comes from these two directions mentioned above. The platform represents capability, and the better the platform, the stronger the capability, and the fewer parts that need to be managed by users. The container represents the deployment pattern, and by extension, the orchestration management, continuous delivery, etc.

The author believes that **the goal pursued by cloud-native, that is, the use of modern container technology, constantly moves the nonfunctional requirements required by the application to the cloud platform and ultimately achieves the goal of allowing developers to focus on the business and implement the business logic.** In essence, the goal of building software is to implement the business

requirements, i.e., the core value of the customers. In order to run and maintain the software and ensure its availability, we have to spend a lot of time building the quality features of the software. And with cloud-native technology, it brings development back to its original intention, allowing developers to focus on the essential issues in software development.

1.2.2 Cloud-Native Technologies

In this section, we will briefly introduce the core cloud-native technologies.

Microservices

The decentralized management, independent deployment, and other features of microservices allow applications to gain great flexibility. And as a booster, the container technique pushes microservice architecture to become more and more popular. Compared with monolithic applications, independent units based on business modules are easier to be containerized and therefore easier to onboard the cloud. On the other hand, microservices collaborate with each other based on APIs and are exposed in an accessible way, which is in line with the “port binding” concept of the 12 factors. These characteristics of the microservice architecture make it more suitable for releasing, deploying, and running on the cloud. This is why the view of industry is that cloud-native applications should be microservices oriented.

Container and Orchestration

The cloud-native ecosystem is gradually growing along with the container and orchestration. Compared with virtual machines, containers have higher efficiency and run speed based on OS isolation mechanism; they can use resources on the cloud with finer granularity, and the lower cost of creation and destruction makes it easier to accomplish the auto-scaling of applications on the cloud. Based on the application organized by containers, it is also easier for us to complete the deployment and distribution. To accomplish the scheduling and management of a large number of containers requires orchestration technology. How to schedule containers, how to manage the container life cycle, and how to maintain the consistency between the real state and the desired state of the application are the core of orchestration. As the container orchestration market stabilized, Kubernetes became the de facto standard in the field, and the entire cloud-native ecosystem was initially built around Kubernetes. So we can assume that with the existing technology tools, containers and their orchestration technology are the cornerstones of cloud-native and unshakeable.

Service Mesh

Service mesh is a new technology that is only been known since 2017. According to the technology maturity curve, I believe that service mesh is still in the middle and early stages of the life cycle. Even so, CNCF has mentioned service mesh in its revised definition of Cloud Native, which shows its importance.

Briefly, service mesh is an infrastructure layer used to manage service-to-service traffic through the capabilities provided by a set of sidecar proxies. If we call Kubernetes the operating system for cloud-native applications, then the service mesh is the networking layer for cloud-native applications, solving traffic management challenges for microservice applications. As the number of services increases, microservices need to be governed and controlled, and the service mesh is the solution used to manage service communication. In addition, the product form of service mesh is based on the sidecar proxy, and the sidecar is the capability provided by container technology.

The service mesh and cloud-native are very similar to vision. Cloud-native wants developers to focus on building applications and own nonfunctional requirements for the cloud platform. The service mesh is a practitioner of this philosophy and implements traffic management into the infrastructure as a sidecar proxy, giving applications the ability to manage traffic, security, and observability, all with complete transparency to the application. Major cloud providers are researching this area with great interest, and open source and hosted products have been launched one after another, and many service mesh products have emerged. Service mesh has changed the way microservice architectures are managed in terms of traffic control, is more advantageous than invasive SDK, and is an important part of cloud-native technologies.

Serverless Computing

Serverless computing refers to running applications in stateless computing containers that are triggered by events, are short-lived, and are used similarly to calling functions.

Serverless computing further refines the granularity of resource usage by applications and can improve the utilization of computing resources. For example, the number of calls to modules in a single application is inconsistent, with some modules being “hot” (more calls) and some being “cold” (fewer calls). In order for these hot modules to support more responses, we had to scale them along with the cold modules. Microservice architecture improves this situation by splitting the hot and cold modules into services so that we only need to scale the hot services. Serverless computing takes this a step further by refining resource utilization down to a smaller granularity. If there is a request, the corresponding compute instance is started; if not, it is destroyed and paid for on demand. From this, the concept of function as a service (FaaS) was defined.

Like service mesh, serverless computing follows the cloud-native philosophy of focusing on the implementation of functionality, not on how to create and destroy the computing resources used, which is the connotation of the serverless computing name. The landscape of CNCF also defined a special module for serverless computing, including related platforms, frameworks, tools, and other ecosystem products.

Immutable Infrastructure

Immutable infrastructure refers to infrastructure instances (e.g., servers, virtual machines, containers, etc.) that are supposed to be read-only once created and must be replaced by creating a new instance if they are to be modified. Conceptually, the biggest difference between mutable and immutable is how the life cycle of an instance is handled, such as creation, modification, and destruction. Earlier when we used mutable infrastructures such as servers, these machines were constantly changing as applications were running, such as modifying configurations, upgrading systems, upgrading applications, etc. Frequent modifications made the server special and difficult to replicate, and once it goes down, we have to take more effort to build a replacement.

The on-demand concept of cloud computing provides the conditions for immutable infrastructures. Immutable infrastructure fits perfectly with the requirement to start when in use and destroy when not in use. With this approach, applications and environments are highly consistent and can be dynamically scaled, and continuous delivery becomes possible. In terms of implementation, we focus on virtual infrastructure in the cloud, packaging and deploying by container technology, automated building and managing version through images, and continuous integration and delivery of applications. Thus, the immutable infrastructure is an important foundation for building an application on the cloud that is elastic and scalable and can cope with rapid changes.

Declarative API

A declarative is a process where we describe a desired outcome and then the system internally implements the functionality to satisfy that desired outcome. The equivalent is imperative, which tells the system how to do something, and declarative, which tells the system what to do.

The most familiar declarative programming language is SQL, which mainly describes what kind of data is wanted, and we don't need to tell SQL how to execute the query engine and how to go to the database to find the data. The declarative API is an important capability of Kubernetes, and the declarative configuration allows us to define the state we expect. If you use the imperative approach, the Kubernetes API server can only handle one request at a time, whereas the declarative API server can handle multiple writing operations at once, with the ability to merge. Because of this,

Kubernetes can complete the process of reconciling from the actual state to the desired state without outside interference, which is the core of Kubernetes orchestration capabilities. Therefore, CNCF specifically emphasizes and includes declarative APIs as a feature of cloud-native.

DevOps and CI/CD

DevOps is an engineering practice that integrates software development and operations to help organizations grow and improve their products faster. With an automated software delivery process, the process of building, testing, and releasing software is faster, more frequent, and more reliable.

In the traditional organizational structure, development and operation are separated, and it is difficult to complete the delivery tasks efficiently in terms of communication and collaboration. Development teams focus on functional requirements and need to deliver new features frequently, while operators focus on nonfunctional requirements, i.e., system reliability, performance, etc. Ops wants to avoid changes as much as possible to reduce release risk, which conflicts with the developers' goal of delivering features frequently. On the other hand, the lack of understanding of the code by the operators affects the correctness of their choice of the runtime environment, while the developers' lack of familiarity with the runtime environment prevents them from making timely adjustments to the code.

DevOps breaks down this organizational barrier and allows the entire process to run at high speed to keep pace with innovation and market changes. Reducing delivery risk by iterating in small steps can facilitate collaboration between development and operations. With continuous integration and continuous delivery practices, the frequency and speed of releases will increase, and product innovation and refinement will occur faster. A key feature of applications on the cloud is their redundancy and constant change, and DevOps and continuous delivery provide a guarantee to cope with rapid change. Therefore it is not too much of a stretch for Pivotal to make this an important feature of cloud-native. Figure 1.4 illustrates how DevOps works.

Fig. 1.4 DevOps stages
(from <https://devopedia.org/devops>)



1.2.3 *Characteristics of Cloud-Native Applications*

First, let's give a simple definition of cloud-native applications.

A microservice application built on containers and deployed on an elastic cloud infrastructure through continuous delivery.

Such a description may be rather one-sided and needs a more detailed explanation. So as with the introduction of the definition of cloud-native, we have listed some core features to help you better understand what a cloud-native application is.

Containerization

Cloud-native applications should be containerized. At this stage, containers may still be the preferred packaging and distribution technology. Compared with virtual machines, deployment with containers is simpler and faster to run and therefore more suitable for cloud environments. Packaging applications into containers and making them independent and autonomous services allow for an independent deployment to meet the needs of evolving applications and also improve resource utilization.

Servitization

Cloud-native applications should be built based on a microservice, or serverless architecture style, i.e., meeting the independence and loose coupling characteristics. Applications are split into services based on business or functional modules, deployed independently of each other, and interact through APIs. This loose coupling can greatly increase the flexibility of the application in implementation and deployment and also fits with the concept of the cloud platform. We can simply assume that one of the most important capabilities of the cloud is the ability to replicate, and service-oriented applications are more easily replicated.

Independent of Infrastructure

Cloud-native applications should be independent of the underlying infrastructure, such as operating systems, servers, etc. The use of system resources should be abstracted, for example, by defining the CPU and memory usage to run programs without limiting the operating system.

Elasticity-Based Cloud Infrastructure

Cloud-native applications should be deployed on elastic cloud infrastructure, public, private, or even hybrid clouds. With the cloud's dynamic scaling capabilities

(replication capabilities), applications can be dynamically adjusted to the load. This is very different from traditional IT applications based on physical machines, which lack the ability to scale quickly and therefore need to assess worst-case scenarios and schedule excessive machine resources in terms of resource usage. Cloud-based infrastructure applications can dynamically allocate resources as needed when it is deployed.

DevOps Workflow

Cloud-native applications should be run using a DevOps approach. Enhance collaboration between development and operations teams through people, processes, and tools to deploy code to production quickly and smoothly. This, of course, also needs continuous integration and continuous delivery practices. These enable teams to release software that iterates more quickly and responds more effectively to customer needs.

Automation Capability

Cloud-native applications need to have the ability to automatically scale and adjust the capacity of instances in real-time based on the workload. This is done by declaratively configuring the number of resources used for each instance and scaling according to the setting. For example, set a CPU threshold of 70%, and when the CPU of the workload exceeds this value, the application automatically scales, makes a copy of the new instance, and automatically loads it into the load balancer.

In addition, the integration process of the application should be automated such that with each code commit, the continuous integration pipeline is triggered, completing tasks such as merging, integration testing, and packaging for the next release. Automation can also be achieved at the deployment level. For example, configure a policy to shift traffic proportionally to the new version of the application at regular intervals; it is called canary releases.

Quick Recovery

Based on container orchestration capabilities and immutable infrastructure, cloud-native applications should have the ability to recover quickly. Workloads can be quickly destroyed and rebuilt in the event of a failure, and applications can be quickly rolled back to the previous version in a failure without impacting users. Then, based on dynamic scheduling capabilities, applications can recover from failures at any time.

The above list of aspects may not be comprehensive, but by introducing these important features, I believe you can have a more concrete understanding of cloud-native applications.

1.3 From Microservices to Cloud-Native

Microservices are an important part of cloud-native technologies, but essentially they are in a different dimension. Microservices are architecture, a style of building applications, while cloud-native is a broader concept that focuses on the operating environment, deployment patterns, tools, ecosystem, and even culture.

Nomicroservice applications can also be deployed in a cloud-based environment but cannot fit the cloud-native concept as well as microservices. Microservice applications can also be deployed in a noncloud environment; however, it cannot earn benefits from the capabilities of the cloud platform. The two are complementary and mutually reinforcing, and the future trend of building applications is inevitably based on microservice architecture to build cloud-native applications.

When microservice applications are migrated to a cloud environment and gradually introduced various cloud-native technologies, a series of changes occur in the way they are built, operated, deployed, and managed. Microservice applications need to integrate and adopt these new technologies. In this section, we will describe how to move from microservices to cloud-native applications.

1.3.1 *Adjustment of Nonfunctional Requirements*

As mentioned earlier, the nonfunctional requirements are the quality attributes of the application, such as stability, scalability, etc., i.e., the nonbusiness logic, also called control logic, that is implemented to achieve these attributes.

Traditional microservice applications that want to get these properties usually have to introduce them as libraries called SDK. Of course, it is also possible to write business logic and control logic together in a more straightforward and primitive way. For example, when accessing an upstream service, we want to implement a retry feature (control logic) to improve the availability of the application (quality attribute). Then a loop statement can be written outside the business code that calls the upstream service, jumping out of the loop when there is a normal return value, and continuing to try when an exception returns until the set number of loops is completed.

As more and more services have been developed, so does the control logic, and we certainly don't want to do this over and over again, so it makes sense to extract it into a library. For example, when developing Java applications using the Spring Framework, you just need to add a @Retryable annotation to the methods you want to retry, as shown in the example below.

```
@Retryable(value = { RemoteServiceNotAvailableException.class }, maxAttempts = 3, backoff = @Backoff(delay = 1000))
public String getBackendResponse(boolean simulatesretry, boolean simulate
    retryfallback);
```

Implementing nonfunctional requirements through libraries is a big step forward, but there are still some limitations, such as it is language bound, which is not appropriate for a heterogeneous microservice application. Also, although the control logic and business logic are separated, they are still essentially coupled and the application needs to package them together for deployment.

The cloud-native architecture will go further in decoupling to make the business logic nonaware and transparent. Let's take service mesh as an example. To implement the above retry feature, the application itself does not need to do any code change; just add the configuration related to retry in the mesh, and the service proxy will automatically retry when the request fails according to this configuration. This approach is completely transparent as there is no need to bind the programming language, or involve any dependency libraries, or add configuration to the application itself. The following code shows how the retry feature can be implemented as a declarative configuration without any change by the application.

```
- route:  
  - destination:  
    host: upstream-service # Route the request to the upstream service  
    retries: # Retry 3 times with a timeout of 2 seconds per retry  
      attempts: 3  
      perTryTimeout: 2s
```

This is the capability of cloud-native technology, the idea of which is to provide nonfunctional requirements to the infrastructure and let the application focus on business logic. Therefore, with the addition of cloud-native, the requirements and implementation of microservice applications in this area need to be adjusted, and the development team should recognize such a change and give full consideration to it during technology selection, and try to implement it in a cloud-native way to reduce development costs. Of course, the original implementation and maintenance costs in this area may be transferred to the infrastructure level, which is also in line with the principle of trade-offs in software development and needs to be considered in the design phase.

1.3.2 *Change in Governance*

A well-running system does not need to be governed; it only needs to be governed when problems arise. In the case of environmental problems, the environment may need to be treated because of pollution or ecological imbalance. Microservices also need to be governed because they have to solve the corresponding problems. For example, services do not know each other and need service registration and service discovery; service performance or invocation anomalies need to be observed; services are overloaded with requests and need traffic management, etc.

The traditional way to achieve governance is by using some microservices framework or SDK, as I mentioned above. Let's also take Spring Cloud as an

example, which is a powerful microservices development framework that provides many components with service governance capabilities, as follows.

- Eureka: Service registration and discovery
- Zuul: request routing and gateway
- Ribbon: Load balancing
- Hystrix: Fusing, fault-tolerant management

The framework can meet almost all of our needs, but essentially it still runs as a library. Similar to the retry example mentioned above, giving the system these governance capabilities requires us to introduce them at the code or configuration level and package them for deployment. There is a studying cost to be able to use these components and some maintenance effort involved. For introducing service governance, the traditional approach is to implement the functionality, integrate (modify the code or configuration), and maintain.

In a cloud-native environment, service governance is owned by the infrastructure as much as possible. For example, once we containerize the application and hand it over to an orchestration tool like Kubernetes to manage, issues related to service registration and discovery are solved. We add an object called Service for each service, which automatically assigns a virtual IP address (VIP) to the application for access within the cluster. Services can be accessed from each other only by the name of the Service object, and service discovery is handled by Kubernetes' DNS service. At the same time, multiple application instances (Pods) can be matched behind the Service object to achieve load balancing with kube-proxy. None of these tasks require any code-level changes to the application; they are transparent to the application. Our focus will also become how to define the Service object, how to expose the service, etc.

Of course, the downside can come in some cases. Applications are partially migrated to the cloud, and these services are handed over to the Kubernetes cluster to manage the need to interact with the legacy systems, and cross-cluster access can bring technical difficulties that require effort to solve.

1.3.3 Deployment and Release Changes

The problems in the software development process do not disappear based on changes in tools, technologies, and architectures. When we migrate to microservice architecture, the appearance of the application becomes a web, so the deployment and release will also become complicated. Each service requires deployment pipelines, monitors, automatic alerts, etc., and there is not a unified packaging method for services implemented through different languages. On the other hand, coordination and version management is also an issue. The dependencies in a microservice application are likely to be presented as a graph, each service has several dependent services, and you need to ensure that there are no circular calls, which means the

dependencies are a directed acyclic graph; otherwise, your deployment process will be struggling.

Version control of services is also particularly important, as you need to roll back in case of failure or implement features like blue-green deployment or canary releases based on versions. As a result, more complex operations and maintenance or more complex application lifecycle management becomes a new challenge.

When we build and maintain a microservice application, there was initially no unified standards to deal with heterogeneous environments, and containerization makes it all easier. One of its key roles is to standardize the packaging and distribution of microservices through a standard image. From a lifecycle management perspective, a cloud-native technology like containers integrates the packaging and deploying of heterogeneous systems, and the differences between services become less and the commonalities become more.

An application with a large number of services needs a centralized platform to unify the management of these services, such as Kubernetes. Storage, compute, and network these resources through Kubernetes for unified abstraction and encapsulation, which can let the microservices that have been unified by the container run directly on the platform. We still need to build monitoring, logging, alerting, and other systems, but a centralized approach allows them to be reused and unified management. With such a platform, operators also no longer need to consider how to reasonably assign a service to a specific computing unit.

Containers and orchestration greatly simplify the lifecycle management of microservices and solve the problem of their own operation and maintenance. Relatively, our focus will also move to the platform level to learn how to deploy and release microservice applications based on the platform.

1.3.4 From Microservice Applications to Cloud-Native Applications

There is no doubt that cloud-native is the current trend of cloud computing, but there are still many technologies to implement or integrate in order to build a cloud-native application or migrate from a traditional microservice application to a cloud-native application, complete with full lifecycle management of development, testing, deployment, and operation and maintenance. Figure 1.5 illustrates the difference between a microservice application and a cloud-native application.

Microservice applications split services based on business modules and implement the overall functionality of the application through the interaction between services. When developing an application, it is important to consider the implementation of nonfunctional requirements in addition to functional requirements. In cloud-native architecture, nonfunctional requirements are usually owned by the infrastructure. That makes developers to focus on the business logic, which is the vision of cloud-native technology that we mentioned. Therefore, from this aspect,

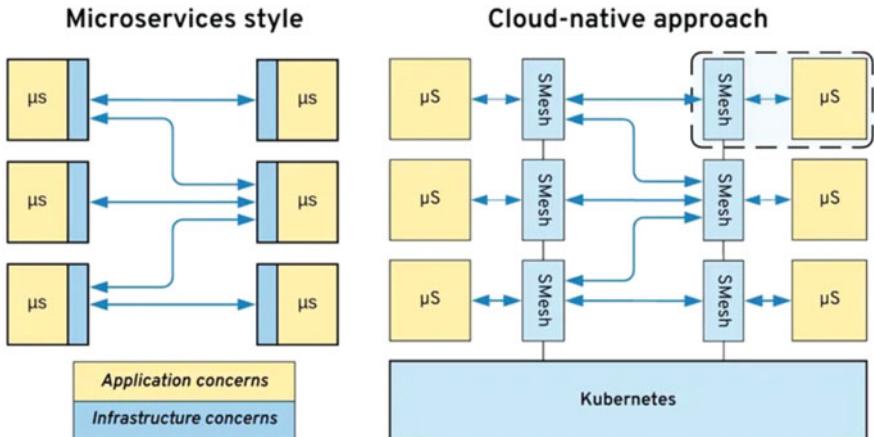


Fig. 1.5 The difference between a microservice application and a cloud-native application (from <https://www.infoq.com/articles/kubernetes-workloads-serverless-era/>)

developers need to have a mindset to build future-oriented cloud-native applications from the concept in terms of architecture design and technology selection.

As for specific technical implementations, we recommend that readers deep dive into the definition of cloud-native, especially the several core technologies highlighted by CNCF.

The first step to migrate from traditional microservice applications to cloud-native applications is to containerize and migrate to a cloud environment so that the applications have the most basic unified operations and maintenance capabilities and gain the scalability provided by the cloud platform. The next step is to use Kubernetes as a infrastructure to implement the orchestration and scheduling management of the application, so that a large number of nonfunctional requirements are stripped out and handed over to Kubernetes. After that, the application has the basic prototype of cloud-native for further evolution around its ecosystem. Then, keep adopting related techniques, such as using service mesh to control traffic, and implementing continuous delivery through DevOps, gradually making the cloud-native features of the application more obvious.

In short, moving from microservices to cloud-native is a gradual process of stripping the nonfunctional requirements of the application and moving them to the infrastructure. During this process, the application should be gradually enhanced and the application lifecycle management should be accomplished using cloud-native technologies, tools, and methods.

1.4 Summary

The modern trend of building applications is “microservices + cloud-native,” which means moving from microservice applications to cloud-native applications.

In order to let the reader understand the core concepts and the relationship between them, this chapter focuses on discussing the relevant theoretical fundamentals. Section 1.1 introduces the concept of microservices and analyzes its core characteristics, and the reader will have a clear understanding and judgment of what a microservice application is. Section 1.2 reviews the definition of cloud-native, explains the core cloud-native technologies, and emphasizes the concept of cloud-native. Section 1.3 combines the characteristics of both, provides a detailed analysis of the key changes and concerns needed to migrate from a traditional microservice application to a cloud-native application, and briefly outlines the basic process of evolving from a microservice to a cloud-native application.

Starting from Chap. 2, we will introduce readers step-by-step how to migrate and build a cloud-native application and share the best practices that we have learned.

Chapter 2

Microservice Application Design



Design is one of the most important activities in the life cycle of software development, whether it be a large complex system or a small feature. Normally, the design process of software engineering includes visualizing ideas by analyzing requirements and possible problems, repeatedly deliberating, making trade-offs, and finally determining a solution. Proper design for a software project is especially important for building robust applications.

In this chapter, we will elaborate on how to design a microservice application based on our team's practical experience. Concretely, we will start from the architectural selection of the application, introduce the solutions of architecture, communication layer, storage layer, and business layer, and also analyze how to transform the legacy system for microservices based on practical cases.

2.1 Application Architecture Design

Architecture describes the way in which an application is broken down into subsystems (or modules) at a higher level and how these subsystems interact with each other. A fundamental issue in the development process is to understand the underlying software architecture, and the overall architecture must be considered prior to which technology or tool to use.

2.1.1 Service Architecture Selection

In an application based on a microservice architecture, each service also needs its own architecture, and in this section, we'll introduce the common architectural patterns.

Demonized Monolithic Applications

Many of the architecture evolution cases we heard started with a critique of monolithic applications: “As applications became more complex, the disadvantages of monoliths came to the fore. So we started work on microservice adoption of applications”.

As the saying goes, if we hear more stories like this, we will invariably have a stereotype of monolithic applications and scoff at the mention of them as if they were backward and full of shortcomings. This perception is very one-sided because we ignore the application scale and organizational structure of these objective factors on the impact of software development.

Figure 2.1 illustrates the impact of monolithic applications and microservices on productivity at different levels of complexity. The horizontal axis represents complexity and the vertical axis indicates productivity. It can be seen that the productivity of monolithic applications is higher than that of microservices when the complexity is low, and only when the complexity gradually increases does the disadvantage of monolithic applications gradually appear and lead to decrease in productivity.

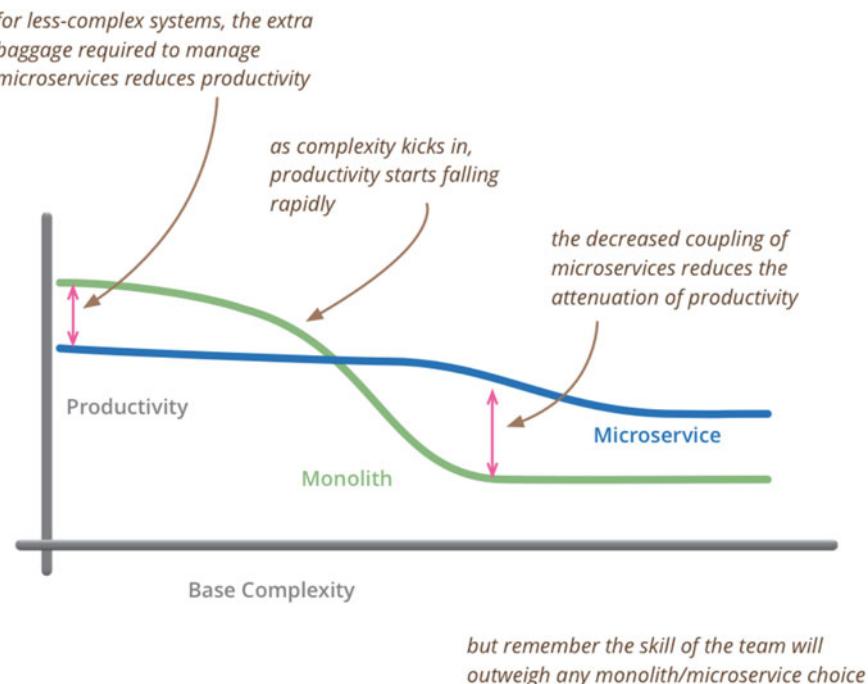


Fig. 2.1 Efficiency comparison between microservice and monolith (<https://martinfowler.com/bliki/MicroservicePremium.html>)

When we start to build an application, how can we determine if it meets the real needs of our customers or if its technology selection makes sense? A good way to do this is to build a prototype first and see how well it works. At this phase, the main considerations are cost and delivery time, and the application needs to get feedback from the customer as soon as possible for quick iteration. In this scenario, a monolithic application is likely to be the most appropriate choice. A monolithic application completes the code from user interface to data access with a single pattern. Initially monolithic applications are considered to be applications that lack modular design, but we can still support the reuse of some of the logic through modular design. A more reasonable way to build the application is to keep the application logically unified as a whole, but make a perfect design of the internal modules, including external interfaces and data storage. This will save a lot of work when transferring to other architectural patterns.

The following benefits can be obtained by using the monolithic application structure.

- Easy to develop: Only a single application needs to be built.
- Easy to test: Developers only need to write some end-to-end test cases and launch the application to call the interface to complete the test.
- Easy to deploy: There are no excessive dependencies, and only the whole application needs to be packaged and deployed to the server.

If the business is simple enough and there is requirement for rapid development, developing a monolithic application is still a reasonable choice, as long as care is taken to design the modules inside the application.

Layered Architecture

The most common design tool used in decomposing complex software systems is layering. There are many examples of layered design, such as the TCP/IP network model. This is generally organized in such a way that the upper layers use the services defined by the lower layers that hide their implementation details from the upper layers.

Layering provides a degree of decoupling capability for applications, with reduced dependencies between layers and the ability to replace specific implementations of a layer relatively easily. Layering also has disadvantages, the first being that too many layers can affect performance, and data usually needs to be encapsulated in a corresponding format as it is passed through each layer. In addition, when the upper layer is modified, it may cause cascading changes. For example, if you want to add a data field to the user interface, you need to make corresponding changes to the query parameter of the storage layer at the same time.

Around the 1990s, a two-tier architecture was a more advanced design, such as the client/server architecture, as shown in Figure 2.2. The typical client/server architecture is the one we know well as Docker.

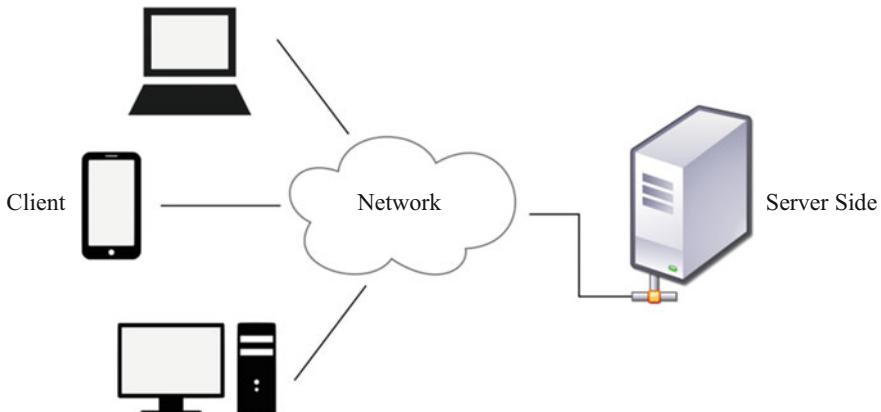


Fig. 2.2 Client-server architecture

One of the problems of using a two-tier structure is where to write the business logic. If the application's data operations are simply to add/delete/modify/query, it makes sense to choose a two-tier structure. Once the business logic becomes complex, it is less appropriate which layer to write this code to. The reason for this problem is the lack of domain modeling.

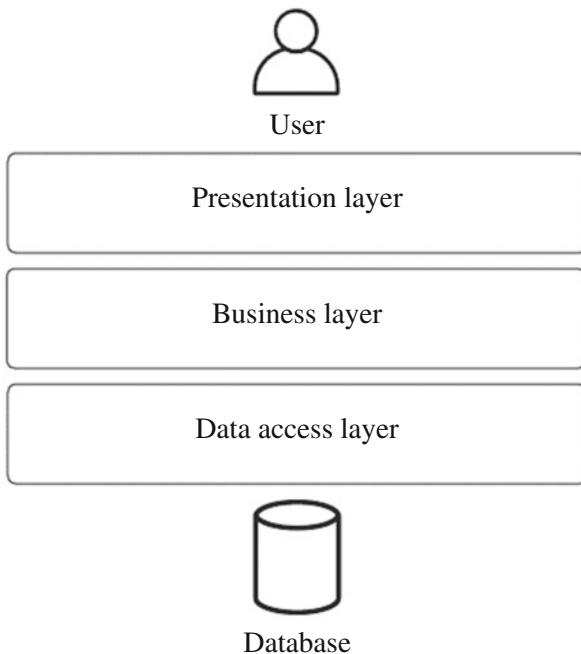
As business changes and grows, the initially designed database tables often fail to accurately present the current business scenarios, and it is a better choice to move to a three-tier structure. The problem of attribution of business logic can be solved by an intermediate layer, which we can call the domain layer or business logic layer.

Based on the three-tier structure, we can describe the application with an abstracted domain model without having to care about the storage and structure of the data, and the database administrator can change the physical deployment of the data without breaking the application. Likewise, the three-tier structure allows the separation of the presentation layer from the business logic layer. The presentation layer is only responsible for interacting with end users and presenting the data returned by the business logic layer to users without performing any operations such as calculations, queries, and updates to the business. The business logic layer, on the other hand, is responsible for the specific business logic and the interface with the display layer, as shown in Fig. 2.3.

If we think of the presentation layer as the frontend (the part related to the web page), then the business logic and data access layers are the backend. The backend can be divided into two layers, but a more common way of layering is to divide the backend into three layers as well, like the following.

- Interface layer or application layer: It is responsible for interfacing with the frontend, completing the coding and decoding work from the frontend request parameters to the business data objects, handling the communication level functions, and invoking the real business logic of the lower layer. The purpose of

Fig. 2.3 Three tier architecture



building this layer is to avoid introducing nonbusiness functions into the domain layer and to ensure the purity of business logic in the domain layer. This layer can usually be built in the Facade pattern so that it can interface with different presentation layers, such as web pages, mobile apps, etc.

- Domain layer: The domain-related business logic, which contains only relatively pure business code.
- Data access layer: Interfaces with data sources to convert business objects into stored data and save them to the database, a process that can be simply understood as "Object Relation Mapping" (ORM).

The disadvantage of a layered structure is that it does not show the fact that the application may have multiple presentation or data access layers. For example, the application supports both web-based and mobile access, but they will both be divided in the same layer. In addition, the business logic layer can be dependent on the data access layer, making the testing difficult.

When discussing a tiered structure, we need to clarify the granularity of the “tiers.” The term “tier” is usually considered to describe layers that are physically isolated, such as client/server. We are talking about the layers represented by “layer,” which is an isolation based on the code level.

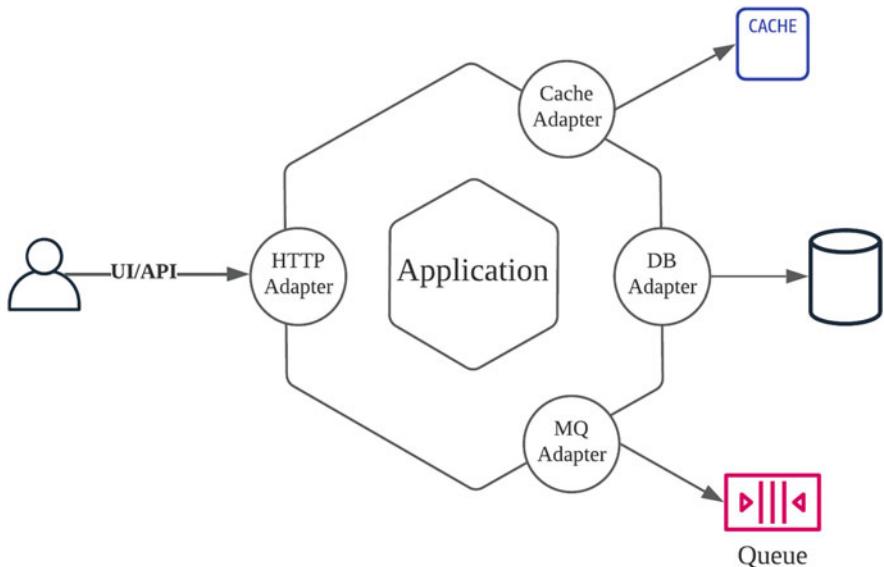


Fig. 2.4 Hexagonal architecture

Hexagonal Architecture

For building a microservice, using a layered architecture is sufficient in most scenarios, but there is a more popular architectural style: hexagonal architecture, also called ports and adapters architecture. It organizes code around business logic, and Fig. 2.4 illustrates the shape of this architecture.

The middle hexagon is the specific business logic, including business rules, domain objects, domain events, etc. This part is the core of the application. On the boundary of the hexagon are inbound and outbound ports, which are usually presented in the form of APIs of some protocols. Corresponding to them are external adapters, which will complete the invocation of external systems and interact with the application through the ports. There are two kinds of adapters, inbound and outbound. Inbound adapters handle requests from the outside by calling inbound ports, such as a controller in the MVC pattern, which defines a set of RESTful APIs, and outbound adapters handle requests from the business logic by calling external systems or services, for example, a DAO object that implements database access or a distributed cache-based client which are typical outbound adapters.

The hexagonal architecture separates the system-level and business-level specific implementations, dividing the entire architecture into two parts.

- System level: The outer boundary of the application, responsible for the interaction with external systems and the implementation of nonbusiness features.
- Business level: It can also be called the domain level, which is the inner boundary of the application and is responsible for the implementation of the core business logic.

Let's use a very common business scenario of take-out food as an example to see how the hexagonal architecture works.

First, the user creates an order through the delivery app on the smart phone, which is located outside the architectural boundary, belongs to the frontend, and sends an order request to the inbound adapter RequestAdapter via some protocol (e.g., gRPC). The adapter is responsible for encapsulating the request input in JSON format into the object needed by the inbound port DeliveryService, and then the DeliveryService calls the delivery of the domain layer to execute the specific business logic. The generated order needs to be saved to the database, which is converted from the domain model to a database relational object by the outbound port DeliveryRepository, and finally the DB adaptor is called to complete the storage. The source code structure can be divided into different directories; examples are as follows.

```
app
├── domain/model
│   ├── Delivery
│   └── DeliveryRepository
└── port/adapter
    ├── RequestAdapter
    ├── DeliveryService
    └── DBAdapter
```

It can be found that the good or bad implementation of the port directly affects the degree of decoupling of the whole application. The inbound port in the above example embodies the idea of encapsulation, which isolates the technical details of frontend request parameters, data conversion, protocols, and other communication layers from the business logic, and also avoids the leakage of the domain model to the outer layer. The outbound port is the embodiment of abstraction, which defines the operation of the domain model as an interface, and the business layer only needs to call the interface without caring about the implementation of specific external services, which also completes decoupling.

The goal of the hexagonal architecture is to create loosely coupled applications that connect the required software environment and infrastructure through ports and adapters. Its main advantage is that the business logic does not depend on adapters, which allows for better separation at the code level and makes the domain boundaries clearer. The concept of hexagonal architecture fits well with the idea of bounded context in domain-driven design and can be used with good results in the splitting and design of services, as described in detail in the book *Implementing Domain-Driven Design*. In addition to this, the hexagonal architecture is much more scalable. For example, if we want to add a new communication protocol or introduce a new database, we only need to implement the corresponding adapter. The

hexagonal architecture is also more friendly for testing support because it is easier to test the business logic by isolating the outer systems. The hexagonal architecture solves the drawbacks of layered architecture and is a better choice for building each service in an application.

2.1.2 *Service Communication Strategy*

To communicate with each other, services must first "know" each other, which is called service discovery. In this section, we will introduce service discovery, service communication, and traffic management based on service mesh.

Service Discovery

Service discovery is a mechanism that automatically monitors and discovers devices or services within a computer network, allowing callers to be dynamically aware of changes in network devices. Probably the largest service discovery system in the real world is DNS, through which we can easily access a huge number of websites on the Internet.

Some readers may say, why do I need service discovery? The application initializes a configuration that contains the addresses of all the services, and then it can access them, right?

Yes, this solution may work if the number of services is small, but for an application that contains hundreds or thousands of services, this solution does not work. First, new services added to the network cannot be automatically discovered by other services, the configuration must be modified, and the application has to restart to load the configuration, and frequent configuration changes will bring higher maintenance costs. In addition, a large-scale service cluster with dynamic node scaling, versioning, and availability status will change frequently, and without an automatic mechanism to detect these changes, there is little way for the system to function properly. Therefore, for microservice architecture, service discovery is very important.

There are usually three roles in a service discovery system as follows.

- Service provider: Can be understood as a web service (protocols can be diverse) that exposes some APIs to the outside and has an IP address and port as the service address. In a microservice application, the service provider can be considered as the upstream service, i.e., the service being invoked.
- Service consumer: Consumes the service and accesses the API of the service provider to obtain data and implement the functionality. In microservice applications, we can think of the service consumer as the downstream service, i.e., the caller.

- Registry: It can be understood as a centralized database to store service address information and is a bridge between service providers and service consumers.

If a service both accesses upstream services and provides APIs to downstream services, then it is both a service provider and a service consumer. There are many services in microservice applications that are like this. There are two patterns of service discovery mechanisms, which we describe in detail below.

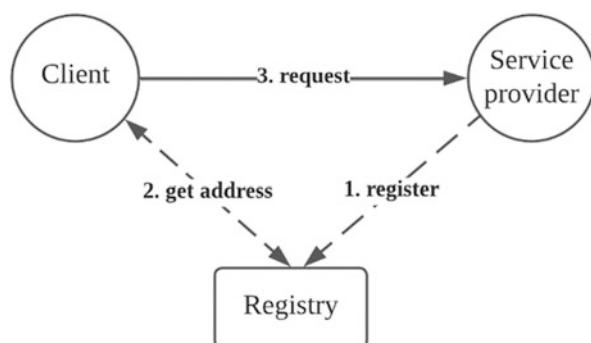
Client Discovery

In this mode, the service will register its address information to the registry when it starts, and the client will visit the registry to get the address of the service to be accessed, and then launch a request to the service provider, as shown in Fig. 2.5.

A typical implementation of the client discovery pattern is Netflix's Eureka, which provides a registry and a client that work together with Netflix's Ribbon component to implement service discovery and load balancing functions. Eureka also has a caching and TLL mechanism, so the data obtained by different services is not necessarily strongly consistent, and it is a more availability-first design. This is easy to understand because this design has met the application scenario of Netflix at that time: the relationship between nodes and services is relatively static, and the IP addresses and port information of nodes are relatively fixed after the services are online, so there are few changes, and the dependence on consistency is reduced.

After the rise of cloud-native technologies such as containers and Kubernetes, this model is somewhat out of step. For example, when containers are scaling up or down, this weakly consistent service discovery mechanism can lead to the inability to sense service address changes in a timely, resulting in access errors. In addition, containerization encapsulates the application well, so we do not need to care too much about the programming language, but the client-side discovery model will have a language-bound SDK, and even involve some language-related dependency packages, which makes the application look a little bloated.

Fig. 2.5 Client-side service discovery



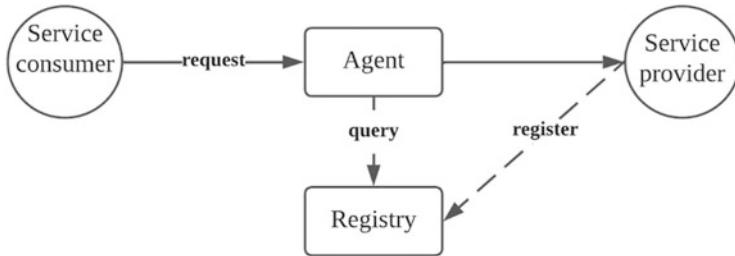


Fig. 2.6 Server-side service discovery

Server-Side Discovery

Compared with the client-side discovery pattern, the most important change in the server-side discovery pattern is the addition of a centralized agent that centralizes the functions of service information pulling and load balancing. The request initiated by the service consumer is taken over by the agent, which queries the address of the service provider from the registry and then sends the request to the service provider based on the address, as shown in Fig. 2.6.

Kubernetes' service discovery mechanism is based on this model. Each Kubernetes node has a proxy named kube-proxy on it, which detects service and port information in real-time. When a change occurs, the kube-proxy modifies the corresponding iptables routing rules at the corresponding node, and the client service can easily access the upstream service by service name. This kube-proxy is the proxy we mentioned above, and it also provides load-balancing capabilities. You can find it in the kube-system namespace of any Kubernetes cluster.

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE	SELECTOR	AGE
kube-proxy	2	2	2	2	2	<none>		6d16h

The server-side discovery pattern has certain advantages over the client-side discovery pattern. It does not require an SDK, hides implementation details, and removes restrictions on languages and libraries.

The downside it brings is that because of the involvement of proxies, requests are forwarded one more time, increasing the latency of the service response. For cloud-native applications built on Kubernetes, using the default service discovery mechanism is the preferred strategy, but if your application is a hybrid deployment, such as having both services inside and outside the Kubernetes cluster, and they also need to interact with each other, then consider an integration solution for service discovery. For example, HashiCorp's Consul project can be integrated with Kubernetes. It provides a Helm Chart that allows you to easily install Consul in a cluster and provides an automatic synchronization mechanism between Kubernetes services and Consul. These features allow applications to work well in cross-cluster or heterogeneous workload environments and provide for easy service-to-service communication.

Service Communication

Microservices architecture builds applications as a set of services and deploys them on multiple instances, which must interact in an out-of-process communication. Therefore, out-of-process communication plays an important role in microservices architecture and there are many technical solutions. For example, one can choose request/response-based communication mechanisms such as HTTP and gRPC; or one can use asynchronous communication mechanisms. The format of messages can be text-based JSON, XML, or binary-based Protocol buffers, Thrift.

Interaction Pattern

Regarding the interaction, we generally consider two dimensions. The first is correspondence, which is divided into one-to-one and one-to-many.

- One-to-one: Client requests are responded to by one service.
- One-to-many: Client requests are responded to by multiple services.

Another dimension to consider is the response method, which is divided into synchronous and asynchronous.

- Synchronization: The client sends a request and waits for a real-time response from the server.
- Asynchronous: The client does not have to wait after sending a request and does not even care about the return result (similar to a notification), and the response from the server can be nonreal time.

The communication mechanism based on the above two dimensions has the following two specific implementations.

- Request/Response: This is the most common communication mechanism, where the client sends a request, the server receives the request, executes the logic, and returns the result. Most of the usage scenarios of a web application are based on this mechanism. Of course, it can also be asynchronous; for example, the client does not need to wait after sending a request, but let the server to callback (callback) way to return the results. This asynchronous callback is generally used when the server is performing a time-consuming task.
- Publish/Subscribe: In fact, it is the observer pattern in the design pattern. The client publishes a message, which is subscribed and responded to by one or more interested services, and the service subscribed to the message is the so-called observer. Publish/subscribe is the most common asynchronous communication mechanism; for example, the service discovery described above is generally based on this mechanism.

Communication Protocols and Formats

Let's look again at the choice of communication protocols and formats during service communication.

- REST/HTTP

REST is a set of architectural design constraints and principles based on the REST style design of the API is called RESTful API. It takes the concept of resources as the core, with the use of HTTP methods, so as to achieve the operation of resource data. For example, the following URL represents the use of HTTP's GET method to obtain order data.

```
GET /orders/{id}
```

RESTful API has many advantages: simple and easy to read, easy to design; easy to test, you can use curl and other commands or tools to test directly; easy to implement, as long as you build a web server can be used; HTTP is widely used, easier to integrate.

Of course, it also has some disadvantages, such as working at layer 7, which requires multiple interactions to establish a connection and slightly inferior performance; it only supports one-to-one communication, so if you want to get multiple resources in a single request, you need to achieve it through API aggregation and other means.

Nevertheless, the RESTful API is still the de facto standard for building web applications, responsible internally for requests and responses from the front and backends, and externally for defining API interfaces for third parties to call.

- RPC

RPC, or remote procedure call, works by using an interface definition language (IDL) to define interfaces and request/response messages and then rendering the corresponding server-side and client-side stub (Stub) programs through which the client can invoke the remote service as if it were a local method.

RPC generally contains transport protocols and serialization protocols; for example, gRPC uses HTTP/2 protocol to transport Protobuf binary data, and Thrift supports a variety of protocols and data formats. The following is an interface to get user information through Protobuf, where the client receives the returned UserResponse by passing in a message body containing the user ID and calling the GetUser interface. Depending on the language, the message body may be rendered as an object or a structure. From this point of view, RPC is better

encapsulated than HTTP and is more in line with the domain-oriented modeling concept.

```
service UserService { rpc GetUser (UserRequest) returns (UserResponse);  
}  
  
message UserRequest {  
    int64 id = 1;  
}  
  
message UserResponse {  
    string name = 1;  
    int32 age = 2;  
}
```

As you can see from the above example, using RPC is also simple and can be considered as not limited by language because of the multi-language SDK support. In addition, RPC usually prefers binary format data for transmission, although the binary data is not readable, the transmission efficiency is higher than the plaintext format data such as HTTP/JSON. RPC communication mechanism is also richer, not only supporting request/response such a way of back and forth, but gRPC also supports stream (Streaming) type transmission.

RPC is more suitable for calls between services, or between systems within an organization. For open API that needs to be exposed to the outside world, it is still preferred to use RESTful APIs for the simple reason that HTTP-based RESTful APIs do not have any requirements on the caller's technique stack, while RPC will create dependencies because it requires a specific IDL language to render the stub code, and the caller needs to introduce the stub in its own application, and in addition, there are also security concerns if the IDL source files are shared with the caller.

In addition to the two protocols mentioned above, there are also TCP-based sockets, SOAP, etc. These protocols have their special usage scenarios, but they are not the first choice for building stateless microservice applications, so they will not be introduced here.

Use Scenarios

The following communication methods and user scenarios are currently included in our application, as shown in Fig. 2.7.

- OpenAPI: Externally exposed RESTful API, based on HTTP, is a typical one-to-one synchronous communication.
- UI interface: Similar to OpenAPI, the frontend UI will call the interface of the backend service through HTTP requests.
- Synchronous communication between services: Within microservice applications, services are invoked via gRPC for better performance.
- Asynchronous communication between services: Services do not always interact with each other using direct invocations, and some scenarios are more suitable for asynchronous event-driven interaction. For example, a user performs a

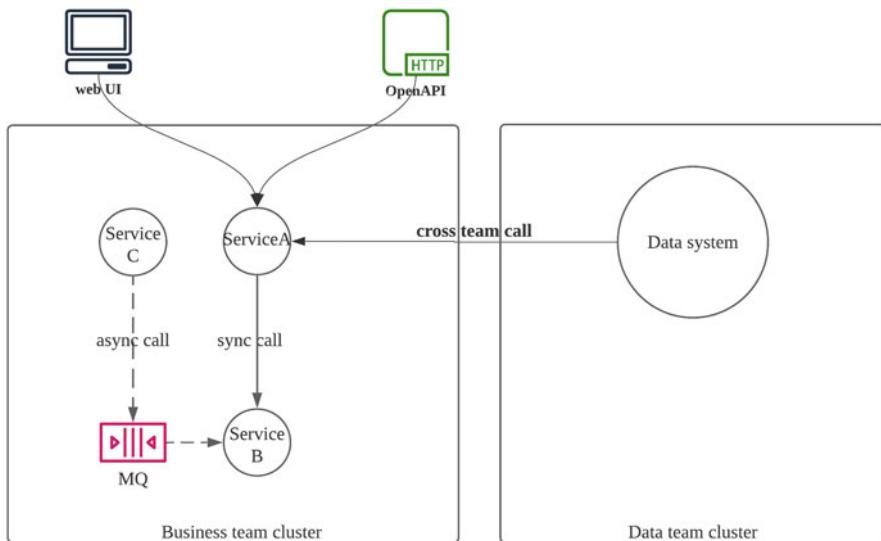


Fig. 2.7 The source of request

forecasting task and sends a task completion message to the message queue when the task is completed.

- Cross-system communication: Our services and other legacy systems also need to interact with each other, and in general both synchronous and asynchronous methods are used. For newer systems, communication is usually still done in the gRPC way. Individual legacy systems, because of the technique stack and other reasons, will use the RESTful API to communicate, and callbacks are used in asynchronous scenarios to communicate.

Traffic Management Based on Service Mesh

Microservice applications typically have three origins of requests: requests from the outside (such as externally provided APIs), requests for access from other legacy systems within the organization, and requests between services within the microservice application. All of the traffic needs to be managed, such as implementing dynamic routing, traffic shifting, or resiliency capabilities like timeout and retry. Some traditional common libraries or frameworks such as Spring Cloud already provide very sophisticated traffic management capabilities, although there are some disadvantages compared to cloud-native solutions like service mesh.

- Language binding: The application needs to involve these traffic management features with SDK or packages, and the language used must be consistent with the framework writing language when heterogeneous microservice applications cannot use a unified solution.

- Coupling: They, although functionally decoupled from the business logic, are not a transparent solution because they bring dependencies on SDK and require additional configuration for use or some code within the application. Also, these dependency packages will be included in the application release package, which is actually coupled at the code level.
- Operation and maintenance costs: These common frameworks are usually deployed as an independent service, with certain labor and resource costs.

Service mesh is a traffic management technology that has become popular in recent years. In short, service mesh is a network infrastructure used to manage service-to-service communication by deploying a sidecar proxy along with each microservice to implement various traffic management features. It has a relatively obvious advantage over a solution like a common framework.

- Transparent to the application: This is the main reason for the popularity of service mesh and its best feature. Service mesh basically works by forwarding requests through a sidecar proxy, which gives the sidecar the ability to act on requests accordingly, such as forwarding different requests to different services based on the request header. It's kind of like an interceptor in a communication framework. And these traffic management capabilities are provided by the sidecar proxy, and applications do not need to modify code or add configuration to access these capabilities, which allows the service mesh to be transparently accessed by applications and is not limited by the development language or technique stack.
- Use in a cloud-native way: Service mesh fully follows the cloud-native concept of decoupling network-related nonfunctional requirements down to the infrastructure layer from applications. At the usage level, service mesh is also used in a cloud-native way like declarative configuration for applications to use traffic management capabilities.

Of course, the service mesh is not a silver bullet, and one of the worrying issues is latency. Because of the sidecar proxy, the service-to-service direct call becomes three calls: from service A to sidecar proxy A, then from sidecar proxy A to sidecar proxy B, and finally from sidecar proxy B to service B. This will certainly increase the latency to some extent. Also because of the increase in the number of forwarding, the debugging difficulty increases accordingly, and it is necessary to use distributed tracing and other features to assist in debugging.

After about 5 years of development, the service mesh technology has gradually matured and become an important technology selection solution for microservice applications in traffic management, and more and more organizations have realized the practice of implementation in the production environment. However, the adoption of new technology will certainly bring certain costs and risks, and its feasibility should be analyzed with the current situation and business characteristics of your team. The author has the following suggestions on whether to use service mesh.

- Requirements for traffic management: If your current application does not have the ability to manage traffic, and you have an urgent need, you can evaluate the service mesh as an important option during technology selection.

- Unification of traffic management capabilities for heterogeneous microservice applications: If your microservice application is a heterogeneous one developed in different languages and cannot use a single framework, and at the same time you want to use a unified solution to implement traffic management, then service mesh is a good choice.
- Pain points of existing solutions: If your application architecture currently has traffic management capabilities, but there are a number of pain points, such as different technique stacks of applications using different implementation methods and cannot be unified, the framework upgrade and maintenance costs are high, the update of the SDK leads to business services also need to update, and so on, and these pain points have brought you long-term trouble and significantly reduce the development efficiency, you can consider the use of service mesh to solve these pain points.
- Technique stack upgrade for legacy systems: If you have a legacy system, suppose it is a monolithic application, large and difficult to maintain, and you are planning to migrate it to a microservice application. This case is perfect for introducing a service mesh. On the one hand, it is cheaper to implement a new architecture based on the new architecture to access the service mesh without considering the compatibility issues of the old system; on the other hand, the application migration is already costly and the additional cost of adopting the service mesh is not worth mentioning.
- Evolution of cloud-native applications: If your team is passionate about technology innovation and wants to build cloud-native applications, then adopting a service mesh will be an important evolutionary process.

Although service mesh has many advantages, I still recommend analyzing it based on your own scenarios and also considering the cost of adoption and subsequent maintenance. Chapter 4 of this book will detail our team's implementation practice of using service mesh technology to manage the traffic of microservice applications.

2.1.3 Storage Layer Design and Selection

The compute-storage architecture proposed by John von Neumann still holds true for microservices: most of the components in the system are stateless, and a few components are responsible for persisting data. The storage layer is usually located at the bottom of the system architecture and can be called the “backend of backend.”

1. Database

When designing the storage layer for microservices, the first question to answer is which database to use. The importance and complexity of the database are evident in the fact that a database administrator position is available in any software company of any size.

The business requirements for the database can be analyzed in the following dimensions.

- Data model: including the number of entity attributes, whether the complexity of the association relationship, query, and write granularity is consistent, etc.
- Data volume: including stock and incremental, sharing, redundant storage, etc.
- Read and write scenarios: including read and write rates and rate of change, concurrency, whether it is sensitive to latency, whether the distribution of read and write data is concentrated (whether there are hot zones), and so on.

To meet these needs, the common database architectures and technologies can be categorized according to the following dimensions:

- OLTP vs OLAP: OLTP (Online Transaction Processing) is for customers with small data input and output levels and low computational burden but is latency sensitive and requires high data accuracy and timeliness; OLAP (Online Analytical Processing) is for data expert customers and company managers with large data input and output levels and high computational burden, which may require PB-level input data. It is relatively insensitive to latency.
- SQL vs NoSQL: SQL is a traditional relational database that supports ACID transactions, represented by MySQL. NoSQL was initially a rebellion against traditional SQL databases and database design paradigms, but as it has grown, NoSQL has been borrowing and integrating SQL design, and even feeding into SQL database design. There is a tendency for the two to go in the same direction. NoSQL can be subdivided into key-value (read and write entities with identical granularity and zero association), search engine (supporting full-text search and compound query based on text analysis), and a graph database (specializing in querying complex relationships and using scenarios such as social network and knowledge graph) (including social networks, knowledge graphs, etc.) and several categories.

Combining the required dimensions and technical dimensions introduced above, microservices can refer to the following decision steps in storage layer design and selection.

- If the requirements of a data model, data volume, and read/write scenarios are not complex, SQL database is preferred.
- Under the typical scenario of OLTP (latency-sensitive and small data volume), SQL database can be given priority for data partitioning (splitting library and table); if the read and write performance of SQL database cannot meet the requirements, middleware such as cache and queue can be considered for introduction.
- OLAP typical scenario (not sensitive to latency, data volume beyond the scope of single-computer bearing, complex model), you can choose the time-series database, Hive, column storage database, etc.
- More complex hybrid scenarios allow different microservices to use different databases and storage middleware according to their needs. For example, using a search engine to support full-text search, using an in-memory

key-value database to store user login information, using a temporal database to record and query user interaction events and data change history, and so on.

2. Caching

Caching is a widespread class of components and design patterns in computer hardware and software systems; typically represented in the storage layer is the in-memory key-value database, which is faster than the database of reading and writing hard disk and can be used as middleware in front of the main database, producing a leverage effect of small hardware to pry large traffic. Although processing data entirely in memory is fast, the cost is that data loss can be caused by machine power failure or process restart. To address the volatility issue, some caches (such as Redis) also offer the option of data persistence.

One of the main design metrics of a cache is the hit ratio. A higher hit ratio indicates that the cache is more useful, and on the other hand, it predicts a greater impact on the database when the cache is hit. Another design metric for caching is the balance of traffic distribution. If a key carries too much traffic (hotspot), it can easily cause damage to the storage hardware (memory, network card), and a hotspot hit may even trigger a new hotspot, causing a system avalanche.

For an application to use the cache, it needs to allow for transient inconsistencies between the data in it and the data in the database. The application can choose to update the cache along with a write to the database (either before or after), or it can choose to update the cache and set the cache expiration time along with a read to the database, or it can even choose to update the cache periodically offline and only read the cache online, without hitting or penetrating to the database. If the application sets a cache expiration time, it should also be careful to handle traffic spikes to the database between cache expiration and cache update.

3. Queues

Queue, as a first-in-first-out (FIFO) data structure, is the basis for the implementation of persistence, cross-node synchronization, and consensus mechanisms for various database systems, which ensures that data entering the database is not lost or out of order. For example, MySQL's Binlog and Lucene's Transaction Log are in fact a kind of queue. As a middleware, a queue can be blocked in front of the main database to solve the problem of high concurrency (merge to string) and dramatic changes in flow rate (peak shaving). In the Event-driven Architecture design, the importance of queues is raised again as the source of truth for the system, solving the synchronization problem of multiple database systems.

The core design metrics for queues are throughput and queue length. The queue length increases when the production capacity is greater than the consumption capacity, and decreases vice versa. If the production capacity is larger than the consumption capacity for a long time, it will break through the queue and cause the system to be unavailable, and then the expansion of the queue (to increase the consumption capacity) needs to be considered. On the contrary, if the production capacity is less than the consumption capacity for a long time, it

will cause the queue resources to be idle, and it is necessary to consider the reuse of the queue (enhance the production capacity).

Due to the serial nature of the queue, the application needs to identify and deal with the problem of a small amount of data limiting the throughput of the queue (analogous to traffic accidents causing road congestion). The solution is usually to set up a reasonable timeout retry mechanism on the consumption side, and to remove the data with more than a certain number of retries from the congestion queue and store it elsewhere, such as another “dead letter” queue.

Microservices have an advantage over traditional monolithic applications in terms of flexibility of storage layer design and selection: different microservices can use different databases and storage middleware. At the same time, microservice architecture introduces many issues specific to distributed systems, such as how to maintain transactional constraints across services, how to deal with data loss, disorder, staleness, and conflicts caused by network delays and outages, and so on.

The design and selection of the storage layer for cloud-native applications is an emerging area with many open issues. For example, in the technology selection, in order to achieve certain business needs, should we directly use the SaaS database of the cloud provider, or based on PaaS hosting open source database, based on IaaS build and operation and maintenance database, or even choose the storage layer without the cloud to ensure the control of sensitive data? In terms of operation and maintenance, how to quickly and correctly scale up and down the storage layer, and whether to use containers and container orchestration like a stateless computing component? In terms of product design, how to avoid or reduce cross-region data read/write and synchronization, and weaken the impact of network latency on user experience? In terms of security, how to design and implement access rights and content resolution for data on the cloud, especially in scenarios where the company and the cloud provider pose a competitive relationship in business?

In short, the storage layer design and selection of data-intensive applications should follow the principles of simplicity, reliability, and scalability: “don’t use a bull knife” to kill a chicken, do not blindly seek new changes, and do not overly rely on ACID transactions, stored procedures, foreign keys, etc.

2.2 Legacy System Modification

The major effort of microservice application development is to refactor an existing system rather than build an entirely new one. So migration is more difficult; you need to choose a reasonable refactoring strategy for quality. In this section, we will introduce the concepts in terms of project implementation: greenfield and brownfield. Then, we will introduce the methods and strategies for migrating monolithic applications into microservices using the strangler pattern.

2.2.1 *Greenfield and Brownfield*

The terms greenfield and brownfield are often mentioned during project execution and are widely used in IT, construction, manufacturing, and other industries. They are related to project development and their specific meanings are listed below.

- Greenfield: A greenfield project means developing a completely new application where we have the freedom to choose the architecture, platform, and all other technologies. The whole development process starts from scratch and requires analyzing requirements, specifying technical solutions, and implementing them one by one.
- Brownfield: A brownfield project is a project where some work has already been done. For the software industry, updating and refactoring existing applications can be considered a brownfield project. We need to analyze the legacy system first, possibly starting with a code review to fully understand the details of implementation. In industry, brownfield usually means waste and pollution, and its color is a good visual representation of this characteristic. Brownfield projects in the software field are not difficult as an industry, but they do face more constraints for the developer taking them on than developing a greenfield project.

Greenfield projects are virtually unlimited in terms of technology and design selection, do not depend on the technology stack of a legacy system, do not require integration with legacy code, and are much freer at the implementation level. Typical greenfield software projects are generally of two types.

- Develop a whole new application that enables a whole new business.
- Rewrite an old system in a new language, implementing only the original business logic, but without using any of the old code.

Developing greenfield projects sounds advantageous and is a generally preferred approach for developers. After all, it's much easier to paint on a blank sheet of paper than to modify someone else's work. However, the disadvantages of this type of project are also more obvious: larger investment, higher costs, and take more time to adopt. Factors that need to be considered include the team, project management, hardware, platform, and infrastructure, not only the development process itself. The scope of the project also needs to be carefully planned to align with business requirements and to avoid mistakes in the direction. Therefore, if your team is limited by cost and time and wants to migrate to a new technology stack quickly, choosing the greenfield approach is not a good fit.

Brownfield projects often benefit from the convenience of legacy code, reducing the cost of developing new projects. However, it can cut down the design flexibility of developers. Many projects are abandoned due to poor design, confusing implementation, or code smell, and developers are reluctant to even touch a single line of code in them, preferring to start over. But clients are more willing to try to refactor rather than overturn based on cost and effort considerations. Of course, not all brownfield projects are useless. As long as the code is well structured, and the

project is easy to maintain, an experienced development team can still take over a brownfield project well. A typical brownfield project generally has the following characteristics.

- Integration of new features into existing systems.
- Modify existing code to meet business changes.
- Optimize the performance of existing systems and improve the quality attributes of applications.

For legacy system enhancement, teams need to make a choice between greenfield and brownfield depending on their situation. If time and cost permit and you want to completely abandon the old technology debt and embrace the new technology stack, you can directly design a greenfield project. For scenarios with limited resources and a desire to gradually migrate to a new technology stack without impact on online business, it makes more sense to choose the brownfield approach.

2.2.2 *Strangler Pattern*

Strangler Pattern is a system reconstruction method named after a plant called strangler fig that Martin Fowler saw in the Australian rainforest. This plant wraps itself around the host tree to absorb nutrients, then slowly grows downward until it takes root in the soil, eventually killing the host tree and leaving it an empty shell, as shown in Fig. 2.8.

In the software development industry, this became a way of rebuilding a system by gradually creating a new system around the old one and allowing it to grow slowly until the legacy system was completely replaced. The advantage of the strangler pattern is that it is a gradual process that allows the old and new systems to coexist, giving the new system time to grow. It also has the advantage of reducing risk. Once the new system doesn't work, you can quickly switch traffic back to the old one. This sounds very similar to a blue-green deployment. Our team gradually migrated monolithic applications into microservices based on the strangler pattern, which smoothly adopted the new technology stack.



Fig. 2.8 Strangler fig migration (from <https://dianadarie.medium.com/the-strangler-fig-migration-pattern-2e20a7350511>)

The Development Process of the Strangler Pattern

Like strangling figs, the strangler pattern has three phases in the development process.

- Convert: Creates a new system.
- Coexistence: Gradually stripping functionality from the old system and implementing it in the new system, using reverse proxies or other routing techniques to redirect existing requests to the new system.
- Delete: When traffic is moved to the new system, the functional modules of the old system are gradually deleted or maintenance is stopped.

Our migration of the legacy system involves both frontend and backend. The backend part is to split the original business logic into microservices one by one; the frontend part is to refactor the original Ruby application into a React.js UI interface and call the new APIs provided by microservices.

The steps and strategies for the migration of the backend part are detailed below.

Identify Business Boundaries

Business boundary is called boundary context in Domain Driven Design (DDD), which divides the application into relatively independent business models based on business scenarios, such as order, payment, etc. Identifying business boundaries is the basis for the reasonable design of microservices, which can achieve high cohesion and low coupling. An unreasonable boundary division can lead to services being constrained during development due to dependencies.

A typical way to identify business boundaries is to use domain-driven design, but it needs to be done by domain experts together with developers who are familiar with DDD. Another simple method is to identify by the existing UI interface because usually different businesses will have their own interfaces. However, this method can only break down the business modules in a relatively rough way, and the detailed parts need further analysis. For example, in our UI page of order module, a function is called “Run Forecast,” which is obviously a forecasting service rather than an order service. Another way is to identify the business boundaries based on the business model and combine the code of the old system. This approach is more suitable for organizations with clear business and a clear division of functions in the team before our team used this approach to identify microservice boundaries and split services.

Rebuilding Based on API

A web application containing front and backends is usually developed around an API interface, so our developing process is also centered around the API. First, we

make a count of the backend APIs called by the frontend pages, set the priority according to the importance, urgency, and impact, and then implement the original interfaces one by one in the new system. One point to note is that if the new interface is to be called by the old page, then it should be consistent with the original interface; if the new interface is only an internal interface or only called by the new frontend page, then certain adjustments can be made based on the characteristics of the new technique stack; just pay attention to the way the exposed page is used and the data format is compatible with the customers.

Select the Least Costly Part for Refactoring

Identifying the business boundary also means that the service unbundling is basically completed and the development of the new system can be carried out. In addition to infrastructure like microservice architecture, migration of business logic should be done based on the principle of least cost, which means that less important business is selected for migration first, such as business with low usage, query-related business, and business with simple logic. Using this strategy can reduce the risk, and even if problems occur, they will not have a big impact on customers. Figure 2.9 shows the process of splitting functions from the legacy system and implementing them in the new system.

Implement the Migrated Functions as Services to Reduce the Modification of the Legacy System

Ideally, in addition to migrating existing services to the new system, new incoming requirements should also be implemented in the new system as much as possible so that the value of the service is captured as quickly as possible and to stop the monolithic application from continuing to get bigger.

But the reality is often less than perfect. The first tricky issue is a timeline, where a customer brings up a new requirement and asks to release it as soon as possible by a certain due date. Developers need to carefully evaluate the effort of implementing it,

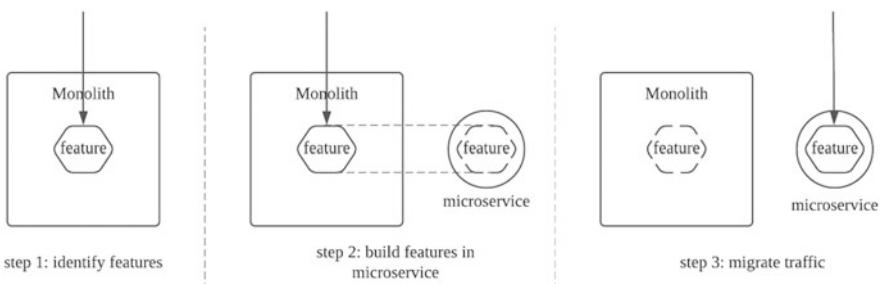


Fig. 2.9 The migration from the legacy system to microservice

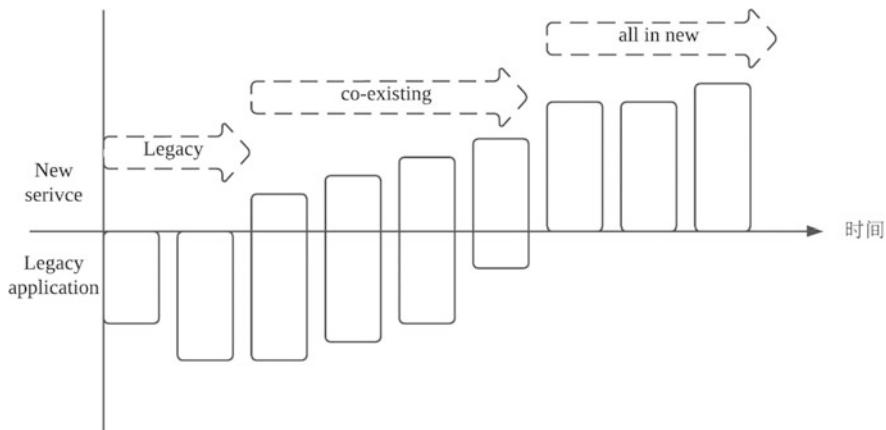


Fig. 2.10 The process of application migration

and if it takes more time to implement it in the new system, then they have to implement it in the old system and then choose the right time to migrate to the new system. This kind of repetitive work is the thing developers never want to do, but they also have to compromise for deadlines. Another situation is that the new feature is heavily coupled with the old system or is temporarily not suitable for separate implementation as a new service. For example, we need to add one or two fields to the original interface, or combine the old code to implement a new logic, etc. It is obvious that the cost of implementing these functions in the old system is much lower, so it is relatively more reasonable to choose to modify the old system when encountering such a scenario. But in any case, when implementing the strangler pattern, developers still need to be clear about this principle: minimize the modification of the legacy application.

Figure 2.10 shows the process of gradually migrating an existing monolithic application into a new microservice application. As you can see from the figure, the old system does not always get smaller because of the reasons mentioned above, it may get bigger. For a certain period of time, the new system and the old system coexist until the new system fully implements the functionality of the old system, and then we can stop maintaining the old system and deprecate it.

The migrating strategy and steps for the frontend part are detailed below.

1. Analysis of UI Interface

Generally, a web application is developed around an API, and the data of the UI interface is obtained from the backend microservices. Therefore, we can find the correspondence between the frontend and backend by analyzing the API's subordination.

In the application, the content of the first-level navigation bar of the UI interface is generally determined based on the large business module, which usually includes several sub-business modules belonging to different microservices. For example, the ad module includes secondary modules such as

advertiser, ad campaign, ad creative, etc., which belong to different microservices. The APIs called by these more clearly defined pages of the business are generally from the same service as well.

However, if there is a situation where data is obtained from other services, you need to pay attention to it. For example, there is a page where the data of charts are obtained from two different APIs and combined together, and these two APIs belong to different services. In such cases, I suggest doing a design review first to analyze whether there is an unreasonable split in the services. If there is no design problem, then the page is responsible for the team belonging to the big business and another team is responsible for providing the API.

2. The Old Frontend Invokes the New Service

For customers, microservice migration is usually a transparent process, and the URLs provided to end-users generally do not change. On the other hand, the frontend pages corresponding to the new service are not yet developed, which requires us to use the old frontend to call the new service. Therefore, after ensuring that the new service has been able to replace some services of the old system, we usually modify the API of the frontend to call the backend and transfer the data acquisition from the old system to the new service.

3. New Frontend Invokes for New Services

This case is easier because the front and backends are rebuilt, and the input and output of the original interface can be optimized and adjusted, as long as the final result shown to the customer is consistent with the original. We defined a URL prefix for the new frontend separately during the migration process, and the rest of the URL remained the same as the original address. The two frontends exist at the same time, and a comparison test can be performed by entering the corresponding URLs to analyze whether the data rendered by the new page is consistent with the old one. We also developed a special comparison tool to highlight the inconsistent parts of the page for easy debugging.

4. OpenAPI Replacement

In principle, the API exposed to the outside cannot be changed casually, and our solution is to release a new version of OpenAPI; for example, the customer used the V3 version of the API before, but now the V4 version of the API based on microservices is released, and the two versions coexist, and the customer is told to migrate to the new version as soon as possible, and the old version will stop being maintained sometimes. Although the new version of the API can use new signatures, such as URLs, and input parameters, unless necessary, the results of the API should preferably be compatible with the old one because customers' applications generally have to do further processing based on the response data, and if the results change, customers will have to modify their code logic, which will increase the cost of migrating them to the new API.

The Strategy of Using the Strangler Pattern

In addition to the migration strategies and steps mentioned above, the following issues need attention during use.

- Don't replace legacy systems with new ones all at once; that's not strangler pattern, and reduce the risk by controlling the scope of the replacement.
- Consider storage and data consistency issues to ensure that both old and new systems can access data resources at the same time.
- Newly built systems should be easy to migrate and replace.
- After the migration is complete, the strangler application either disappears or evolves into an adapter for the legacy system.
- Ensure that the strangler application is free of single points and performance issues.

Applicability of the Strangler Pattern

Strangler pattern is also not a silver bullet and does not work for all system migration scenarios. There are some conditions and limitations to using it, as follows.

- Web or API-based applications: A prerequisite for implementing the strangler pattern is that there must be a way to switch between the old and new systems. Web applications or applications built on APIs can choose which parts of the system are implemented in which way through the URL structure. In contrast, rich client applications or mobile applications are not suitable for this approach because they do not necessarily have the ability to separate applications.
- Standardized URLs: In general, web applications are implemented using some general patterns like MVC, separation of presentation, and business layers. But there are some cases where it is not so easy to use the strangler pattern. For example, there is an intermediate layer below the request layer that does operations such as API aggregation, which results in the decision to switch routes not being implemented at the top layer, but at a deeper layer of the application, when it is more difficult to use the strangler pattern.
- Smaller systems with less complexity and size: The strangler model is not suitable for rebuilding smaller systems because the complexity of replacement is lower and it is better to develop completely new systems directly.
- To systems where requests to the backend cannot be intercepted: Requests cannot be intercepted means there is no way to separate the frontend and backend, and there is no way to direct some of the requests to the new system, and the strangler pattern is not available.

2.3 Business Logic Design

The core of an enterprise application is the business logic that enables the business value of the customer. Business logic is distributed across multiple services, making it more challenging to implement a complete business workflow. Properly splitting services can reduce the difficulty of implementation and avoid unreasonable

invocation. In this section, we will focus on how to split services and how to design API interfaces that conform to the specification.

2.3.1 *Splitting Services*

Splitting services is the hard part of the design phase, and you need to know the application very well and then abstract the domain model and follow design principles in the splitting process. Let's go through them one by one.

Understanding the Application

An in-depth understanding of the application you want to split is a prerequisite for splitting the service. You need to understand the business requirements, user scenarios, system operation mechanisms, legacy code, and many other aspects. Our team's approach is to review the requirements and design documents and compare the code of legacy systems to understand the implementation details. Understanding the requirements and user scenarios allows you to deepen your understanding of the business in order to subsequently abstract a reasonable domain model. Reading the code allows you to review the previous design and understand the technical details and possible constraints. It is important to note that the purpose of reading old code is to add details to the requirements, never be imprisoned by old code in a design mindset. The original implementation is likely to be inapplicable in a microservice architecture, and there may need to be optimized that should be refactored.

The first step is to identify the system behaviors, which can usually be analyzed by verbs in user stories and scenarios. These actions will be the basis for designing the API interface. For example, in the scenario of a user creating an order, the action “create” is an action that will eventually be designed as an interface like `CreateOrder`.

For more complex scenarios, more effective tools are needed to assist in the analysis. Our team uses event storming for business analysis. It is a lightweight and easy-to-master domain-driven design approach. Event storming is useful to help project participants have a unified view of the business, such as key business processes, business rules, business state changes, user behavior, etc. It helps developers to sort out the business, discover domain models and aggregations, and, to some extent, separate the boundaries of the business to prepare for splitting service.

Event storming is usually run through a workshop; find a meeting room with a whiteboard, prepare various colors of stickers, and you are ready to go. You can also use online collaboration tools for materials that are not easily prepared. For example, we started with stickers, but found that moving and adjusting them was a bit of a hassle when we had more stickers, so we chose to use Lucidspark to do it online, and it was much more efficient. The content about event storming is not the focus of this book, so we will not repeat it here. Interested readers can go to the official website to

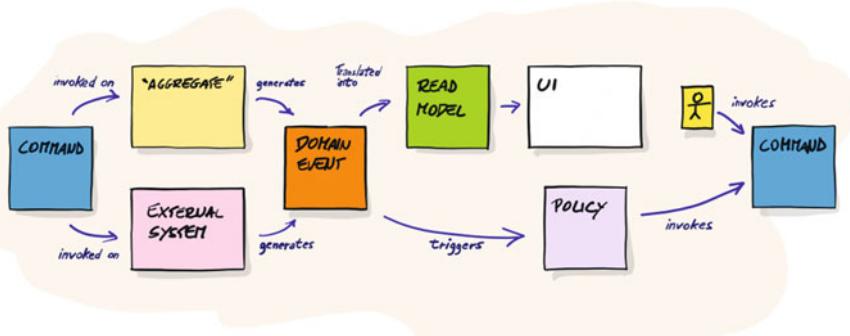


Fig. 2.11 The model of event storming (from www.eventstorming.com)

understand the implementation details. Figure 2.11 shows the basic model of event storming.

Split According to Business Capacity

Business usually refers to business activities that can generate value for a company or organization and its specific capabilities depending on the type of business. For example, the business of e-commerce applications includes product management, order management, delivery management, etc. The business of the advertising platform we have developed mainly includes customer management, advertising management, placement management, etc.

The first step in splitting services is to first identify these business capabilities, which are generally known through the organization's goals, structure, and business process analysis. The following is a list of business capabilities that a takeaway system might have.

- Merchant management
- Consumer management
- Order management
- Delivery management
- Other

After identifying the business capabilities, the next thing to do is to map the business to services. It is important to note that business capabilities are usually hierarchical, i.e., a business contains multiple sub-branches. For example, the order business can be divided into many parts such as creating orders, accepting an order, food pickup, and delivery. Therefore, which level of business to map as a service requires specific analysis. I suggest following the principle of high cohesion and low coupling: belonging to the same large business area, but relatively independent of each other, can be split into different services; conversely, the top business with very

single capability without subdivision items can be defined as a service. For example, there is no shortage of payment processes in order processing, and order information and payment are obviously very different, so they can be defined as order service and payment service, respectively. Another example is that the business of ad placement has sub-parts such as ad campaigns, ad inventory, and placement bids, which are both interactive and relatively independent, and are suitable to be split into multiple services.

The benefit of splitting services based on business capabilities is that they are easy to analyze and the business is relatively stable, so the whole architecture is more stable. However, this does not mean that services will not change. With business changes and even technical trade-offs, the relationship between services and services may be reorganized. For example, if the complexity of a business grows more and more, the corresponding service needs to be decomposed, or too much frequent communication between services will lead to low performance, combining services together is a better choice.

Split According to the Domain

The concept of Domain Driven Design (DDD) has been around for almost 20 years, and microservice architecture has given it a second life, making it a very popular design approach. The reason is simple: microservice architectures are well suited to design using DDD, and the concept of subdomains, bounded contexts, can be a perfect match for microservices. Subdomains also provide a great way to split services. A subdomain is a separate domain model that describes the problematic subdomains of an application. Identifying subdomains is similar to identifying business capabilities: analyze the business and understand the different problem domains in the business, define the boundaries of the domain model, i.e., the bounded context, and each subproblem is a subdomain corresponding to a service. Take also the take-out case as an example; it involves commodity service, order service, and delivery service, including commodity management, order management, delivery management, and other problem subdomains, as shown in Fig. 2.12.

Splitting a service using a domain-driven approach generally involves the following two steps.

Business Scenario Transformation into the Domain Model

This step requires abstracting the domain model and dividing the bounded context based on the relationship between the models, i.e., a set of highly cohesive business models with clear boundaries. It can be considered that this is the microservice we want to define.

Our team uses the event storming mentioned above to identify domain models. This approach starts from the perspective of system events, takes the events generated by the change of system state as the focal point, then connects the events in the

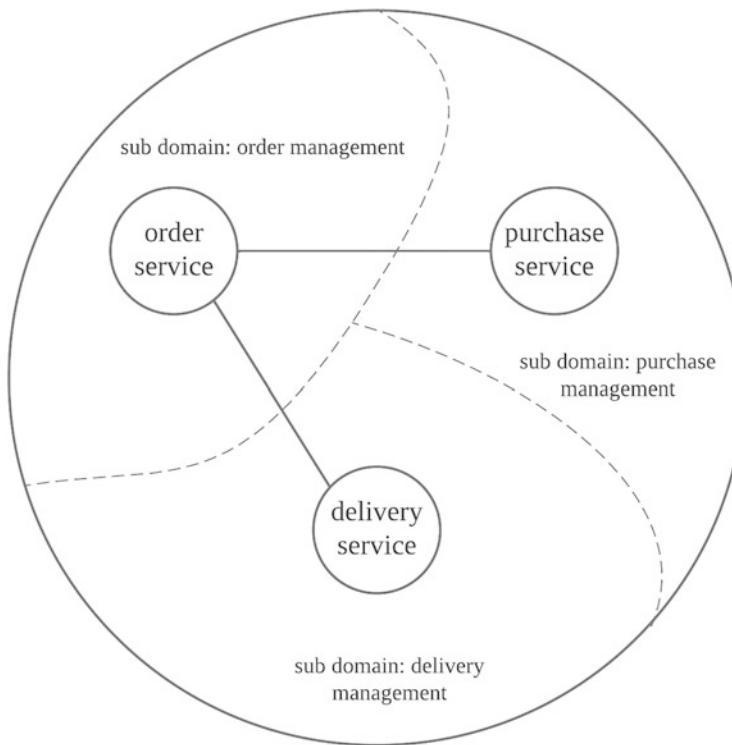


Fig. 2.12 Sub domain of the service

whole business process according to the chronological order, analyzes the actors, commands, read models, policies, etc. related to the events, finally identifies the aggregators and bounded contexts. For example, the events in the core business process of the take-out system include orders created, user paid, merchants accepted, take-out dispatched, etc. The actors corresponding to these events are consumers, merchants, dispatchers, etc. Through the analysis, we can also clearly identify the events corresponding to the domain model of orders, payments, merchants, users, etc., and then define the aggregation according to the interaction of the model; the whole process of business modeling is basically completed.

The analysis process is inevitably biased. For the identified domain models, the reasonableness of the model can be judged based on the principle of low coupling and high cohesion. For example, are the dependencies between the subdomains divided as few as possible? Are there any unreasonable interdependencies? In addition, you can take a quick look at the business process mapping to the model in your mind to see if the model can meet the needs of the business, and think about whether the model can be easily extended when the business changes. Of course, we still have to emphasize that design often has to compromise for reality, and we usually consider performance, complexity, cost, and other factors and find an acceptable design balance.

Convert Domain Model to Microservices

After analyzing the domain model, then we can design services, system architecture, interfaces, etc. based on it. These are typical software design aspects and the reader can design based on their own habits using appropriate methods, such as agile methods. For microservice architecture, two other design aspects need to be taken into account.

- Service dependencies: A reasonable microservice architecture should have fewer dependencies. The first thing we need to be careful of is that there should not be a situation where two services depend on each other, which is commonly referred to as circular references. A reasonable dependency topology should be a tree or a forest, or a directed acyclic graph. The service provider does not need to know who invoked it, i.e., the upstream service does not need to know about the downstream service, so as to ensure that the system is locally dependent and not globally dependent. Globally dependent applications do not satisfy the microservice architecture requirements but are so-called distributed monolithic applications.
- Service interaction: The split microservices cannot hold business features independently and must interact with each other. In the choice of interaction methods, we should pay attention to the user scenarios; in general, there are the following interaction methods.
 - RPC remote process invocation: RPC is the author's preferred method of interaction between services; communication is relatively more efficient and easy to integrate. gRPC generates some additional dependencies and requires the generation of the corresponding server and client according to the interface description language (IDL) used. Our microservice application uses the gRPC invocation method.
 - HTTP/JSON invocation: Communication by defining a RESTful API in HTTP with data in JSON format. Its biggest advantage is that it is lightweight and the coupling between services is extremely low, and only the interface needs to be known to invoke it, but it is usually slightly inferior to RPC in terms of transmission performance.
 - Asynchronous message invocation: This is the “publish-subscribe” interaction. This approach is ideal for systems based on event-driven design. The service producer publishes a message, and the listening party consumes the message and completes its business logic. The message interaction eliminates the dependencies associated with imperative invocations and makes the system more resilient.

It is not necessary to choose only one way of service interaction; it makes more sense to use different ways for different scenarios. Readers are free to choose according to their own situation. The diagram of the above two steps is shown in Fig. 2.13.

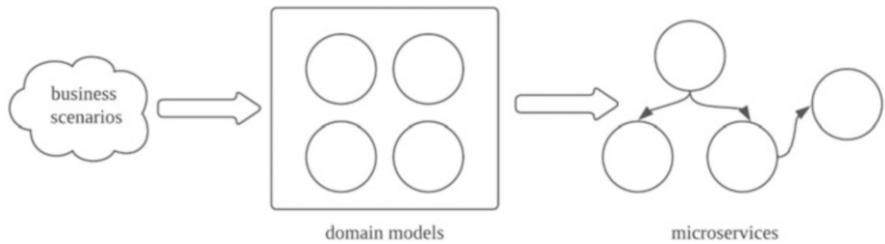


Fig. 2.13 The steps of building service

Domain-driven design is a relatively complex design approach, and it may take a book to explain it clearly. Here we only do a simple guide, teams with experience in domain-driven design can try to use it to complete service splitting, and the final results and the results based on business capabilities should be consistent.

Splitting Principles

The split service needs to be small enough to allow small teams to develop and test it. But how small is the right scope of the service? This is often an empirical conclusion, especially since the adjective “micro” can be somewhat misleading. We can draw on some object-oriented design principles that are still relevant to defining services.

The first is the Single Responsibility Principle (SRP). In object-oriented design, the single responsibility principle requires that there is generally only one reason to change a class: the defined class should have a single responsibility, responsible for only one thing. Defining a service is actually the same, that is, to ensure that the service is focused on a single business, to achieve a high cohesion, the benefit of doing so is to improve the stability of the service.

Another principle that can be referred to in service splitting is the Closed Package Principle (CCP). It originally meant that the changes made to a package should all be within the package. This means that if an update causes several different classes to be modified, then those classes should be placed within the same package. In the event of a change, we only need to make the change to one package. In a microservice architecture, the same parts that are changed for the same reason need to be placed in the same service, reducing dependencies and making changes and deployments easier. For a particular business rule change, as few services as possible should be modified, ideally only one service.

In addition to the two principles mentioned above, after the service split, we can then check whether the service design is reasonable based on the following rules.

- Services must be cohesive and should implement strongly related functions inside the service.
- Services should adhere to the principle of public closures, while changes should be put together to ensure that only one service is affected by each change.

- Services should be loosely coupled, with each service well encapsulated, exposing the APIs that provide business capabilities to the outside, and can be modified internally without affecting external access.
- The service should be testable.
- Each service should be small enough to be developed and completed by a small team.
- The service development team should be autonomous, i.e., able to develop and deploy its own services with minimal collaboration or dependency.

Difficulties of Service Splitting

There is no perfect solution in software design; more often than not, it is a compromise with reality. There are times when there is no way to guarantee a perfect split, such as the following common difficulties and problems that may be encountered.

- Excessive cross-process communication: Service splitting can lead to networking problems. The first is that cross-process communication causes network latency, which is more pronounced when the amount of data transferred is relatively large. In addition, if the business invocation link is long, for example, it takes several services across to complete a business process, this inevitably reduces the availability of the application. The more nodes are intervened, the higher the probability of problems. To cope with this problem, on the one hand, it is necessary to introduce such capabilities as distributed tracing in the application so that the root cause can be traced quickly when problems occur; on the other hand, service governance should also become an indispensable capability in the architecture, and it is a good choice to introduce governance capabilities in a transparent way using a service mesh, which we will describe in Chapter 4.
- Distributed transactions: One of the data-level pain points faced by migrating monolithic applications into microservice architectures is distributed transactions. The traditional solution is to use a mechanism like a two-phase commit, but this is not appropriate for web applications with high concurrency and traffic. A more common approach in the Internet industry is to use compensation that prioritizes the performance of the application and ensures the eventual consistency of the data. In Chapter 5, we will present a distributed transaction solution implemented by our team based on Sagas theory. Besides, it is also an idea to eliminate distributed transactions by some sequential adjustments in the business process.
- The god class is difficult to split: In the field of object-oriented programming, a god class is a class with too much responsibility, where much of the application's functionality is written into a single "know-it-all" object that maintains most of the information and provides most of the methods for manipulating the data. Therefore, the object holds too much data, has too much responsibility, and acts as a god. All other objects depend on the god class and get information. Splitting the god class is especially difficult because it is over-referenced and holds data from many different domains. Domain-driven design provides a better way to

split the god class by implementing different versions of the god class for the respective domain models that cover only the data and responsibilities within their own domain.

2.3.2 *Design API*

Once the splitting service is done, the next step is to map the system behaviors to the services, i.e., to design the API. If a transaction needs multiple services to work together, the API defined for service-to-service communication we generally call an internal API. There are usually three steps in designing an API: determining the resources to be operated, determining how to operate the resources, and defining the specific details, which we describe below.

Determine the Resources to Be Operated

A resource is some kind of data model in a business system. The APIs for manipulating resources are usually RESTful APIs, which are the de facto standard under HTTP/JSON format, and the OpenAPI exposed to the public and the interfaces called by the frontend follow this. REST is a resource-centric API design style, where clients access and manipulate network resources through Uniform Resource Identifiers (URIs) and determine how to operate on the resources through a set of methods, such as get, create, etc.

When using HTTP as the transport protocol, resource names are mapped to URLs and methods are mapped to HTTP method names. For the internal interfaces that services call each other, we follow the gRPC standard format for defining them. Essentially, the gRPC API is defined in a style very similar to the REST style, which also combines both resource and method parts into interface names. Very often, there is a one-to-one correspondence between resources and the domain model we define, but in some cases, we may need to design split or aggregated APIs, which are analyzed in the context of specific requirements.

A resource is a business entity that must have a resource name as a unique identifier, typically consisting of the resource's own ID and the parent resource ID, etc. To take the simplest example, we want to define an API for the user management module, and the core resource is the user. The URIs for obtaining resources are generally expressed in plural form, which is also known as a collection. A collection is also a resource, meaning a list of resources of the same type. For example, a collection of users can be defined as users. The following example describes how to define a URI to get a user resource.

```
/users/{user_id}
```

If the resource contains sub-resources, then the sub-resource name is represented by following the parent resource with the sub-resource. The following example

defines the URI to get the user's address, which is a child resource of the user, and a user will have multiple addresses.

```
/users/{user-id}/addresses/{address_id}
```

Determine the Method of Operation of the Resource

A method is an operation on a resource. The vast majority of resources have what we often call add, delete, and get methods, and these are the standard methods. If these methods do not represent the behavior of the system, we can also customize the methods.

Standard Method

There are five standard methods, Get, Get List, Create, Update, and Delete.

- Get

The Get method takes the resource ID as input and returns the corresponding resource. The following example shows the implementation of the Get Order Information API.

```
rpc GetOrder(GetOrderRequest) returns (GetOrderResponse) {
    option (google.api.http) = {
        get: "/v1/orders/{order_id}"
    };
}
```

- List

The list takes the collection name as the input parameter and returns the collection of resources matching the input, i.e., a list of data of the same type is queried. Getting a list is not quite the same as getting a batch, as the data to be queried in a batch does not necessarily belong to the same collection, so the input reference should be designed as multiple resource IDs, and the response is a list of resources corresponding to these IDs. Another point to note is that the list API should usually implement the pagination to avoid returning too large a data set to put pressure on the service. Another common function of lists is to sort the results. The following is the process of defining a Protobuf for a list API, with the corresponding RESTful API defined in the get field.

```
rpc ListOrders(ListOrdersRequest) returns (ListOrdersResponse) {
    option (google.api.http) = {
        get: "/v1/orders"
    };
}
```

- Create

The create method requires the necessary data of the resource as the request body, sends the request with HTTP POST method, and returns the newly created resource. There is a design that returns only the resource ID, but the author recommends returning the complete data, which can help to obtain the data of the fields not sent in the input reference that are automatically generated by the backend, avoiding subsequent queries again and the semantics of the API are more standardized. It is also important to note that if the request entry can contain a resource ID, it means that the storage corresponding to the resource is designed to “write ID” instead of “auto-generated ID.” Also, if the creation fails due to a duplicate unique field, such as an existing resource name, then this should be explicitly stated in the API error message. The following example shows the implementation of the Create Order API; unlike the List, the HTTP method is POST and has a request body.

```
rpc CreateOrder(CreateOrderRequest) returns (Order) {
    option (google.api.http) = {
        post: "/v1/orders"
        body: "order"
    };
}
```

- Update

Update is more similar to create, except that the ID of the resource to be modified needs to be explicitly defined in the input parameters, and the response is the updated resource. There are two HTTP methods corresponding to update; if it is a partial update, use the PATCH method; if it is a complete update, use the PUT method. The author does not recommend a complete update because of the incompatibility problem that will occur after adding the new resource field. Also, if the update fails because the resource ID does not exist, an error should be returned explicitly. The following example shows the implementation of the Update Order API, which is very similar to the Create API.

```
rpc UpdateOrder(UpdateOrderRequest) returns (Order) {
    option (google.api.http) = {
        patch: "/v1/orders"
        body: "order"
    };
}
```

- Delete

The delete method takes the resource ID as input and uses the HTTP DELETE method, which generally returns empty content. However, if the resource is simply marked as deleted and the actual data still exists, the resource data should be returned. Deletion should be an idempotent operation, i.e., there is no difference

Table 2.1 Standard methods definition

Standard methods	HTTP methods	HTTP request body	HTTP response body
List	GET	None	Resource List
Get	GET	None	Resources
Create	POST	Resources	Resources
Update	PUT/PATCH	Resources	Resources
Delete	DELETE	/	/

between multiple deletions and one deletion. Subsequent invalid deletions should ideally return errors not found by the resource to avoid sending repeated meaningless requests. The following example shows how the delete order API is implemented.

```
rpc DeleteOrder(DeleteOrderRequest) returns (google.protobuf.Empty) {
    option (google.api.http) = {
        delete: "/v1/orders/{order_id}"
    };
}
```

Table 2.1 describes the mapping relationships between the standard and HTTP methods.

Custom Methods

If the standard methods described above do not express the functionality you want to design, you can customize the methods. Custom methods can operate on resources or collections, and there are not many requirements for requests and return values. In general, the resources are defined, and the so-called custom methods just define the operation; for example, ExecJob represents the execution of a certain task. Custom methods generally use the POST method of HTTP because it is the most generic; the input information is placed in the request body. The query type operation can use the GET method. The design of the URL varies, and it is generally recommended to use the format “resource:operation,” as shown in the following example.

<https://service name/v1/resource name:operation>

The reason for not using a slash is that this could potentially break the semantics of REST or create conflicts with other URLs, so it is recommended to use a colon or an HTTP-supported character for the split. The following code is a custom API for the cancel order operation.

```
rpc CancelOrder(CancelOrderRequest) returns (CancelOrderResponse) {
    option (google.api.http) = {
        post: "/order:cancel" body: "*"
    };
}
```

Define Specific Details

Once the API signature is defined, it is time to define the specific details based on business requirements, including the request body, the data item of the returned resource, and the corresponding type. For Protobuf, both the request and response are defined as message objects. We continue to use the above example to define request messages for the Get Order and Create Order APIs, respectively. It is important to note that if you subsequently need to add fields to the message, the number after the original field cannot be changed. Because the data transfer format of the gRPC protocol is binary, the number represents a specific location, and modifying it will cause data parsing errors. The new field can use an incremental number.

```
// Get the request message for the order
message GetOrderRequest{
    int64 order_id = 1;
}
// Request message for order creation
message CreateOrderRequest{
    string name = 1;
    int64 product_id = 2;
    int64 count = 3;
    int64 customer_id = 4;
    //...
}
//Order Message
message Order{
    int64 order_id = 1;
    string name = 2;
    int64 product_id = 3;
    int64 count = 4;
    int64 customer_id = 5;
    //...
}
```

Correspondingly, if the interface needs to be exposed to the public as OpenAPI, it is sufficient to define the input and return values in accordance with HTTP requirements, and the request and response bodies generally use JSON format data.

Finally, a few more design considerations. The first is the naming convention. In order to make the API easier to understand and use, the naming generally follows the principles of simplicity, straightforwardness, and consistency. The author lists a few suggestions for reference.

- Use the correct English words.
- Common terms are used in abbreviated form, such as HTTP.
- Keep the definitions consistent and use the same name for the same operation or resource to avoid duality.
- Avoid conflicts with keywords in the programming language.

In terms of error handling, different response status codes should be used to identify errors. One design approach is to return a normal 200 status code for all

requests and define the error field as true or false in the return value to distinguish between the success and failure of the request, which is not desirable unless there is a specific reason why this must be done. Business errors require explicitly informing the caller what the error is, i.e., returning a business error message. Our practical experience is that business errors return a 500 status code with the reason for the error in the error-message field.

In short, defining an easy-to-read and easy-to-use API is not an easy task. In addition to the above, attention needs to be paid to API documentation and comments, version control, compatibility, etc. I suggest the team to define a complete API design specification based on their own situation and follow it strictly.

2.4 Summary

For a complex distributed system like microservices, the design is the first priority. In this chapter, we choose three aspects of application architecture design, legacy system migration, and business logic design as representations to introduce the methods and strategies of microservice design.

At the architectural level, we described several architectures for microservices and how to complete the technology selection for the presentation and storage layers. The most common situation is refactoring a legacy system rather than rebuilding an all-new one, so in Sect. 2.2 we describe how to migrate monolithic applications to a microservice architecture through the strangler pattern. We discussed the two main issues which developers are concerned about, splitting services and designing APIs.

Design ability is a developer's essential skill, which requires long-term accumulation and practice, and the design methodology is too sophisticated to be exhaustive. This chapter focuses on a few core issues to discuss and provide readers with some design ideas.

Chapter 3

Service Development and Operation



In software development, implementation is a very important phase, which refers to the process of transforming software design into software products. In recent years, along with the growth of microservice in the cloud-native era, software implementation is facing new challenges. What development process to choose? How to manage the code? How to build microservices? What is the middle platform strategy? This chapter will introduce our team's practice around service development and operation.

3.1 Agile Software Development

A software development process includes requirements analysis, design, implementation, testing, deployment, and maintenance. The Waterfall Model, a traditional software development standard, requires that these steps be performed linear sequentially to ensure software quality.

In today's Internet era, the traditional Waterfall Model can no longer keep pace with the digital transformation; thus, Agile Development was born. In 2001, 17 anarchist developers gathered in Utah, USA, and after several days of discussion, released the "Agile Software Development Manifesto," announcing the arrival of the Agile movement.

We are uncovering better ways of developing software by doing it and helping others do it. Through this work, we have come to value the following:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

The Agile Development Process is a collection of methodologies that conform to the Agile Software Development Manifesto. Different methodologies have different focuses; some focus on workflow, some on development practices, and some on requirements analysis. The Agile Development Process also includes a series of practices applied to the software development lifecycle, such as continuous integration, test-driven design, and pair programming. The microservice architecture in the cloud-native era is characterized by service componentization, decentralized governance, infrastructure automation, design evolution, etc., which makes an efficient and mature agile development process even more necessary. For this reason, this section will introduce how to implement agile development.

3.1.1 From Waterfall Model to Agile Development

The waterfall model and agile development are top development methodologies. This section will explain why we choose agile software development by comparing these two development methodologies.

Waterfall Model

The waterfall model is a classic software development standard, first applied in manufacturing, construction, and other traditional industrial fields, used to implement such as Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), Office Automation (OA), Office Automation (OA), etc. The waterfall model divides the software development life cycle into six phases: requirements analysis, system design, development and implementation, testing and integration, release and deployment, and operation and maintenance. It requires that each phase must be executed linearly sequentially, and each phase must have a clear output. In addition, each phase can only be executed once and can only occur after the previous phase has been executed. The waterfall model is a typical plan-driven process that requires gathering as many requirements as possible early in the project, planning, and allocating resources. In short, the waterfall model has a detailed plan, clear requirements boundaries, and a clear division of labor among the team.

Agile Development

Agile development is designed to help teams deliver business requirements faster. Agile development requires that the software development process be iterative and incremental. Each iteration cycle is 1–4 weeks. Requirements are confirmed with the product team at the beginning of the iteration to determine the iteration goals. A

stable version needs to be available at the end of the iteration and presented to the product team. In addition, agile development also introduces various practices such as code refactoring, design patterns, pair programming, test-driven development, and continuous integration to improve the quality of the product. The process of agile development is shown in Fig. 3.1.

Agile development has many advantages over the traditional waterfall model:

- Agile development is an iterative development approach with short development cycles, fast iterations, and embracing changes; on the other hand, the waterfall model is a sequential development approach with fixed development cycles.
- Requirements gathering and analysis in agile development are iterative, and communication with customers is continuous, but requirements analysis in the waterfall model is one-time and feedback-free.
- Testing in agile development can occur in parallel with or even before development; on the other hand, testing in the waterfall model can only occur after development is complete.
- Agile development requires only the necessary documentation; on the other hand, the waterfall model requires large quantities of documentation.

Many companies that have followed agile development have achieved real results. However, with the growing size and business complexity, these agile pioneers are more or less facing the lack of stability and auto-scaling brought by

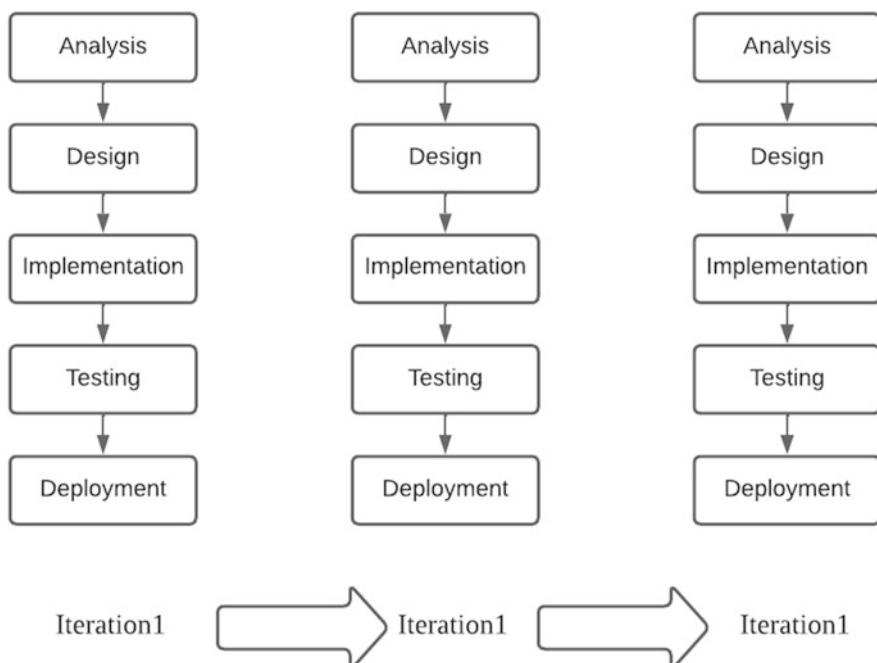


Fig. 3.1 Process of agile development

monolithic applications. Amazon, Netflix, SoundCloud, and others embraced microservices when they reached a certain size and encountered architectural bottlenecks. Microservices are also becoming an important tool for building agile architectures in the development process.

Microservice architecture is characterized by service componentization, decentralized governance, and design evolution. The adoption of the Scrum development process meets the demand of rapid iteration and solves the team autonomy problem brought by service autonomy so that it can solve many challenges of microservices.

3.1.2 *Scrum in Practice*

Scrum is a lightweight agile development framework that balances planning and flexibility, encouraging teams to be flexible in their approach to problems and to learn from their iterations to continuously improve. Jeff Sutherland and Ken Schwaber, the fathers of Scrum, gave the values of Scrum as follows.

Successful use of Scrum depends on people becoming more proficient in living five values: Commitment, Focus, Openness, Respect, and Courage.

Typically, a Scrum team has less than 10 members, and the team delivers projects in iterations, or Sprints, typically 2–4 weeks.

Scrum teams use a Product Backlog to manage product requirements. A backlog is a list of requirements that are prioritized in terms of customer value. The backlog is typically organized as items, each of which can be a new feature, an improvement to an existing feature, or a bug. High-priority items are described in detail.

At the beginning of each Sprint, team members sit down for a Sprint Planning meeting to discuss the work to be performed, select the appropriate items for the current iteration cycle, and define the Sprint Backlog and the team's Sprint goals.

Daily Scrums are held during development, i.e., daily stand-up meetings where team developers review what they have done the previous day and make adjustments based on the Sprint goals. Sprint Review and Sprint Retrospective meetings are held at the end of each Sprint to review the results of the current Sprint, reflect on the successes and failures of the Sprint, and make continuous improvements in the next iteration cycle. In addition, Scrum Scrum, which is a complement to Scrum, can effectively solve the problem of cooperation between different Scrum teams.

The flow of the Scrum is shown in Fig. 3.2.

During the daily project development, we used Jira, a project and transaction tracking tool from Atlassian. It's a commercial platform widely used for project management, requirements gathering, agile development, defect tracking, etc. We used Jira's Scrum Board, Scrum Backlog, Sprint Burndown Chart, and other agile management features. Readers can choose the right agile management platform according to the situation of the companies.

SCRUM FRAMEWORK

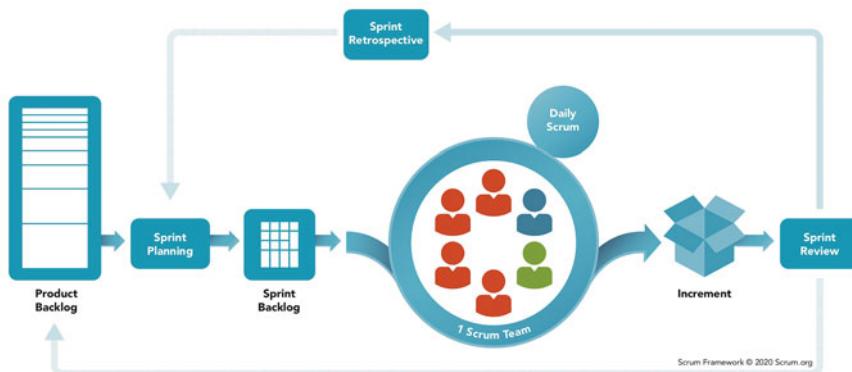


Fig. 3.2 Flow of the Scrum (Figure from <https://www.scrum.org/resources/scrum-framework-poster>)

In addition, using Scrum Framework does not mean that the theory should be rigidly applied to the microservice development process; instead, the theory should be combined with real practice.

The traditional waterfall model fails to embrace changes and neglects communication with end users, but it still has advantages as a classic software development standard. For example, the phases of development, documentation, and others pursued by the waterfall model help junior members of the teams to grow. On the other hand, although Scrum defines roles and events, it does not define how team members should use them. That's why Scrum's best practices are so important.

Below we will provide an introduction to Scrum in practice.

What Is Sprint

Sprint is an iteration cycle, usually 2–4 weeks. The length of a Sprint varies from Scrum team to Scrum team, but for the same team, a Sprint should be a fixed length. A Sprint is the heartbeat of Scrum, and within a Sprint, team members should work toward the same Sprint goal. During this time, Sprint goals cannot be changed at will, but the Sprint backlog can be adjusted and refined if needed.

To ensure the successful completion of Sprint goals, we used the Sprint Burndown Chart provided by Jira for forecasting. Figure 3.3 shows an example of Jira's Sprint Burndown Chart. A Burndown chart is a graphical representation of the remaining workload, with the horizontal axis used to represent time and the vertical axis used to represent the workload. By comparing the ideal remaining work curve

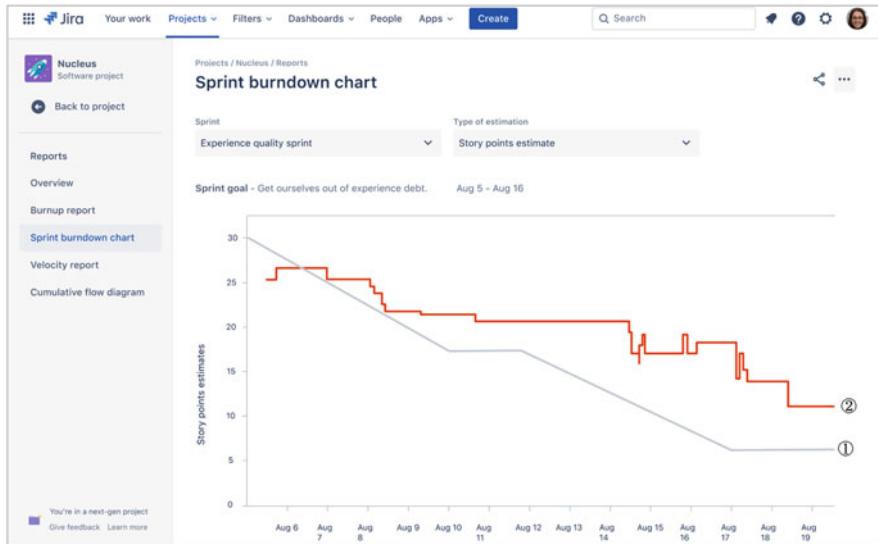


Fig. 3.3 Jira's sprint burndown chart (Figure from Jira)

(curve ①) with the actual remaining work curve (curve ②), you can see how well the Sprint is being completed. If the actual remaining workload curve is much higher than the ideal remaining workload curve, it means that there is a risk of failure within the Sprint and needs to be adjusted in time, while on the other hand, it means that the tasks are overestimated and the team may finish the tasks ahead of schedule. And the judgment made by team members based on experience is also important.

Sprint Planning

The Sprint planning meeting requires all Scrum team members to attend. The goal is to develop a teamwork plan. Attendees are expected to discuss and answer the following three questions.

Question 1

What is the current value?

The value of the Sprint is the Sprint goal, and the entire team should establish the same Sprint goal that can be used to inform project stakeholders why the Sprint is valuable. The Sprint goal should be finalized during the planning meeting.

Question 2

What can be done within the current Sprint?

The Scrum team determines the current Sprint Backlog list by selecting several items from the Product Backlog. During this process, the team's developers can further validate the requirements and thus refine these entries.

Question 3

How do I complete the work within the Sprint?

Each team member should be responsible for a certain number of items to be developed. To achieve this, a workload estimate for each item is required. In addition, team members should further split the items to be developed into individual development tasks, such as front-end tasks, back-end tasks, database design, etc.

“User stories” and “Planning poker” help in the evaluation of the backlog. Each item in the backlog is called a user story. The user story requires that the requirements be described from the user’s perspective and that the value of the story is reflected. In this way, the backlog describes the product requirements and values the user experience. The backlog and user stories can be managed and tracked using Jira. Plan Poker is a method for estimating the workload of user stories. For each user story, participants select the number they think is appropriate from playing cards and show it at the same time. The numbers are generally from the Fibonacci sequence, i.e., 1, 2, 3, 5, 8, 13 Game participants, especially the person who shows the largest and smallest number need to explain the reason. Then team members need to replay until everyone’s numbers are relatively close.

Daily Scrum

In rugby, the players line up before kickoff; the purpose is to get the players familiar with each other on the field. The daily stand-up meeting, which serves a similar purpose as the pre-game parade in rugby, enables team members to get to know each other, thus making Sprint transparent and open. Each team member is informed of both the progress of the Sprint and the progress of development tasks and can also ask for help. Daily stand-up meetings should generally last no more than 15 minutes. The content can depend on the team but generally fall into three categories.

- What did I do yesterday?
- What do I plan to do today?
- Any problem?

The progress of the task can be seen in these questions. The Scrum team can then identify problems and try to solve them after the meeting. In addition, this mandatory sharing can also motivate each team member and improve team cohesion.

Instead of using stickers on the whiteboard for updates, we used Jira’s Scrum Board. As shown in Fig. 3.4, TO DO means not yet started, IN PROGRESS means in progress, and DONE means completed. The Scrum Board can reflect the progress of team members’ work. In addition, a Sprint Burndown Chart can be used to follow the progress of the entire team. By using a mature commercial agile management platform, the team can effectively manage each step of the agile development process and also effectively improve the completion efficiency of Scrum events.

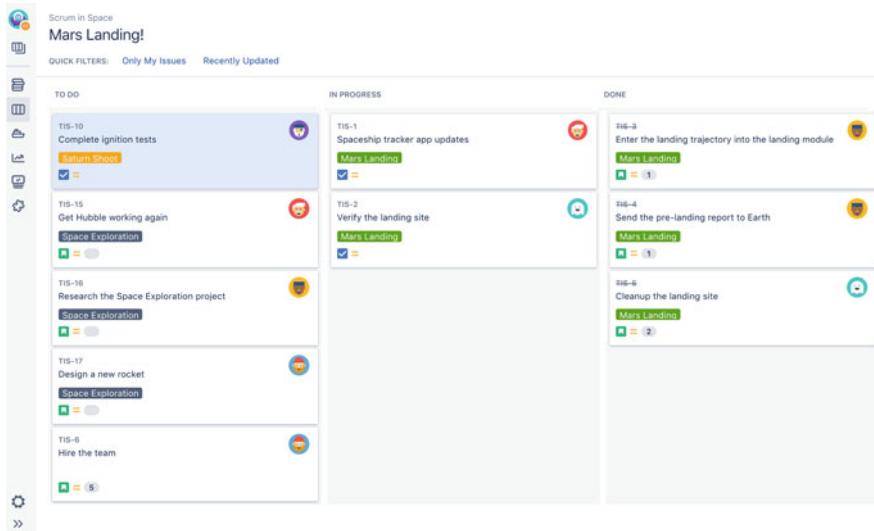


Fig. 3.4 Jira's scrum board (Figure from Jira)

Daily Development

In each iteration cycle, team members need to complete development, testing, and sometimes deployment to achieve Sprint goals. Scrum introduced daily stand-up meetings but did not define daily development practices. We adopted an agile development-based approach to daily development, supplemented by the waterfall model.

We introduce more agile practices besides Scrum. We choose pair programming for better coding. Two developers can work on one computer, one types the code, and the other reviews the code, and they can also swap roles. Pair programming can improve the quality of the code. Besides, the usage of continuous integration and automated testing can ensure product quality. Continuous integration can find bugs early and keep the latest code in a workable state by continuously building and testing. Readers can see more about continuous integration and continuous deployment in Chap. 9 of this book.

On the other hand, the team should not completely abandon documentation. We recommend that team members write design documentation and testing documentation. If the development task is light enough, team members can simply describe the development design and test cases directly on Jira's user stories; otherwise, team members should write documentation and link it with the user stories, and senior team members are obligated to review the documentation.

Sprint Review

The purpose of the review meeting is to demonstrate the results of the entire team's work within a Sprint. Stakeholders of the project should also be invited to the review meeting. Team members can demo runnable outputs at the meeting and should also discuss what was done during the iteration cycle, thus avoiding turning the review meeting into a showcase meeting. As the penultimate event in the Sprint, the review meeting should be a means for the team to celebrate its success.

Sprint Retrospective

The retrospective meeting occurs before the end of the Sprint. It's a summary of the current iteration cycle, requiring all team members to participate in the discussion. The questions discussed generally fall into three categories.

- What did we do well?
- What didn't we do well?
- What else to be improved?

By discussing the above questions, the team should find ways to improve quality and effectiveness and apply them to future Sprints. A review using stickers on a whiteboard can prevent team members from influencing each other. The stickers should be divided into three colors, corresponding to the three types of questions mentioned above, as shown in Fig. 3.5 (the color distinction cannot be reflected in this book, so you should pay attention to it in practice). Team members write the questions on the stickers and sort them on the whiteboard. Eventually, the team should discuss each of the questions on the stickers and translate the results into action items that will guide future iteration cycles.



Fig. 3.5 Sprint retrospective

Scrum of Scrum

Scrum recommends a team of fewer than 10 people, but large enterprise projects often require dozens of people, so it is inevitable to introduce multiple Scrum teams to collaborate. The Scrum of Scrum is designed to solve problems among multiple Scrum teams. Scrum of Scrum can be held from time to time, and each Scrum team needs to elect a representative to participate. The representative does not have to be a Scrum leader or product owner but has to have a deep understanding of the cross-team issues that arise in the project. The representative needs to answer the following questions.

- What affects other teams?
- Does my team help from other teams?
- Any risk?

Scrum of Scrum is a complement to Scrum and can help organizations implement agile development at a larger scale.

Scrum-based agile development practice is a lightweight agile development practice. It aims to transform the traditional software development process into a rapid iterative agile development process while motivating team members and promoting the iterative growth of the team. Thus it's better to introduce agile development practices for microservice architecture, which is also an agile software architecture.

3.2 Runtime Environment

A runtime environment is the hardware and software environment required to run a software product. A software product often requires multiple runtime environments. Different runtime environments apply to different scenarios. Generally speaking, runtime environments are divided into development, testing, staging, and production environments. This section will introduce these four types of runtime environments.

3.2.1 *Development Environment*

The development environment for microservices is the local environment. For example, when developing in the Golang language, it is easy to compile the binaries locally and run them. However, in actual project development, a software product often needs even dozens of microservices. Different microservices often require different external dependencies, different data sources, and programming languages. If developers still try to run these microservices locally, then they need to know the detail about each microservice. It can make the daily work very painful.

We can use container technology to solve this problem. Docker provides Docker Compose, a tool to run multiple containers. It allows developers to define service containers in a docker-compose.yaml file and execute the following command in the path of the YAML file command to automatically build the image and start the service's containers.

```
docker-compose up -d // run in the background
```

Docker Compose solves the problem of starting multiple services, but maintaining YAML files is still painful. If developers want to switch between multiple microservices, they need to keep modifying the docker-compose.yaml file. For this reason, we have developed Docker Switch, a tool for building Docker Compose-based development environments.

Docker Switch contains a docker-compose.yaml file and a docker-switch.json template file. Docker-compose.yaml includes several service images. Docker-switch.json.template is used to generate a template file for local configuration.

To use it, you need to create a new local configuration file docker-switch.json. Docker-switch.json consists of two parts: envs and switches. The former describes the container network, database, and other external dependencies, and the latter specifies which services need to be run in Docker and which services need to be run directly locally. The example is as follows.

```
{
  "envs": {
    "docker_subnet": "172.30.0.0/16",
    "docker_ip": "172.30.0.1",
    "mysql_ip": "host.docker.internal",
    "mysql_port": "3306",
    ...
  },
  "switches": {
    "order": {
      "service": "order",
      "port": "3100",
      "image": "order/order",
      "use": "docker" // 在Docker中运行
    },
    ...
  }
}
```

The startup script will load the docker-switch.json file, generate a .env file to store basic information about the running services, generate a .stopservices file to describe the services that need to be stopped, and generate a .scaleservices file to describe the services that need to be horizontally scaled. The startup script also contains the Docker Compose start/stop commands, as follows.

```
docker-compose stop $(cat .stop_services)
docker-compose up -d $(cat .scale_services)
```

In this way, the developer only needs to modify the docker-switch.json file and restart Docker Switch to complete the service switching between the container and local environments.

3.2.2 *Test Environment*

A test environment is used to test software products. Unlike the development environment, the test environment is usually not built locally but in separate machines. Because microservices and containers have a natural affinity, the test environment should be built in Docker. There are two types of testing environments, regression test environment for individual microservices and end-to-end testing environment for testing from front-end to back-end.

Regression Test Environment

Regression testing can be built into the continuous integration process. Readers can see more about continuous integration and continuous deployment in Chap. 9 of this book. Our team uses a Jenkins-based continuous integration and continuous deployment solution. The merge of code to the master branch triggers a Jenkins pipeline that pulls up instances of the cloud to complete the continuous integration of microservices.

Take Golang language as an example, the Jenkins pipeline will check out the remote code in the cloud compute instance using the Git command, start the dependency services using the docker-compose command, build the microservice binaries using the go build command, run the binaries to start the microservice, and then use the regression test framework to run the regression cases. After all, cases are passed, use Dockerfile to build the service image and push it to the private image repository.

End-to-End Test Environment

End-to-end testing has a different purpose than regression testing. The former is functional testing from front-end to back-end, while the latter is testing for a specific microservice and is the foundation for the former. Therefore, end-to-end testing can be run periodically instead of in a continuous integration or continuous deployment process. Our team also uses Jenkins to build an end-to-end testing environment and uses Jenkins' build periodically feature to build timed tasks.

The Jenkins pipeline will check out the remote code in the cloud compute instance using the Git command, start the dependency services using the docker-compose command, and finally, run end-to-end test cases and generate test reports using the end-to-end testing framework.

3.2.3 Staging Environment

The staging environment is a mirror environment of the production environment and is the last line of defense for the quality of the product. It should be as consistent as possible with the production environment in terms of servers, configurations, databases, etc. Our team will perform different types of testing in the staging environment.

- Manual regression testing before releasing to the production environment.
- Database change testing.
- Some automated tests, such as Performance Test, Post Check, and Chaos Engineering.
- User acceptance testing was initiated by the product team.
- Integration testing of customer systems.

Our team uses Amazon Elastic Kubernetes Service (Amazon EKS) to host microservice clusters. The Jenkins deployment task pulls service images from a private image repository to update the application with a rolling update strategy. Unlike the continuous deployment process for test environments, the deployment of staging environments is tightly controlled. We require that each deployment of a staging environment be documented and tracked.

The database in the staging environment should be separated from the database in the production environment to prevent problems in the staging environment from affecting the production environment. In addition, the data in the production environment database can be copied to the staging environment database before each major release, thus reducing the cost of integration testing with customer systems in the staging environment.

3.2.4 Production Environment

A production environment is used directly by the customers of a software product. The deployment of the production environment is sensitive and rigorous. Canary deployment can be used at this stage for final testing and validation.

Similar to the staging environment, our team uses EKS to host a microservice cluster for the production environment, using continuous integration and continuous deployment based on Jenkins. The Jenkins deployment tasks are passed in the image versions used by the staging environment. We require that each deployment of the

production environment be documented and the issues encountered during the upgrade be collected. Post-check after release is also mandatory.

3.3 Code Management

In recent years, as the scale of software development expands, more and more companies put the improvement of R&D effectiveness on the agenda, in which code management occupies a pivotal position. Code is the most important output of developers, and failure to manage code effectively will inevitably affect productivity. This section will talk about the techniques and practices related to code management so that you can have a clear understanding of how to manage code.

3.3.1 *Git Branch Management*

Code management has evolved from centralized source code management to distributed source code management. Subversion is a centralized version control system, where the repository is centrally located on a central server. Git is a distributed version control system, where the repository is located on each developer's computer, thus eliminating the need for a central server and a network connection for committing code. The latter has become the de facto standard for code management and can deal with all kinds of problems that can arise from team collaboration.

Branch management is the core of Git code management. Proper branch management can help us collaborate efficiently on development. So, what are the common Git branch management strategies? And how do you choose one?

Gitflow Workflow

In his 2010 blog post “A successful Git branching model,” Vincent Driessen proposed a Git-based branching strategy that has been the de facto standard for Git branch management for quite a long time. It defines five types of branches.

- Master branch: A branch that Git automatically creates. There is only one master branch of code, and any official release should be based on the master branch.
- Develop branch: Used to store stable versions of daily development code. Both the develop branch and the master branch are long-standing branches.
- Feature branch: A branch where developers develop a specific feature. The feature branch should be created from the develop branch and merged back into the develop branch when development is complete.

- Release branch: The branch used to do pre-release before the official release and should work with the staging environment. Each time the product is pre-released, the pre-release branch should be pulled from the develop branch for packaging

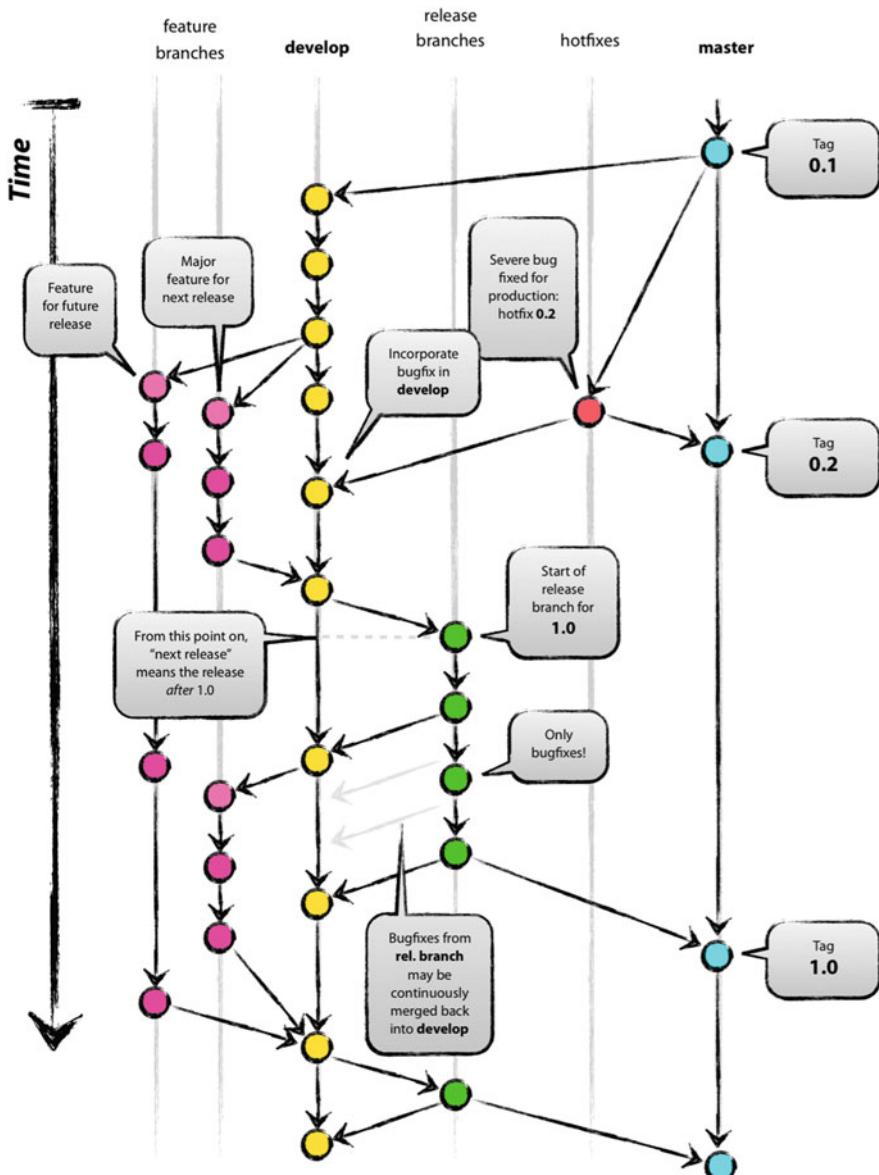


Fig. 3.6 Workflow of the Gitflow (Figure from <https://nvie.com/posts/a-successful-git-branching-model/>)

and deployment, and then merged back into the develop branch and the master branch when the product is officially released. Unlike the feature branch, the pre-release branch should have a fixed command rule, usually denoted by release-*. * indicates the next version number of the official release, e.g., release-3.1.1.

- Hotfix branch: A branch pulled from the master branch instead of the develop branch. Hotfix branches are used to fix bugs in the production environment, so they are created on demand and do not have a fixed version number, and need to be merged back into the develop and master branches after the official release. The hotfix branch is deleted after the merge is complete.

The workflow of Gitflow is shown in Fig. 3.6.

We can divide the five types of branches into two categories: long-term branches and short-term branches. The master and develop branches are long-term branches, and the feature, release, and hotfix branches are short-term branches. The branches that developers use daily are mainly short-term branches. In the following, we will introduce three ways of using short-term branches.

Feature Branches

First, create a feature branch using the following command.

```
git checkout -b newfeature develop
```

Then, merge them into the develop branch when development is complete.

```
git checkout develop  
git merge --no-ff newfeature
```

Finally, use the following command to delete the function branch.

```
git branch -d newfeature.
```

Release Branch

First, create a release branch using the following command.

```
git checkout -b release-3.1.1 develop
```

Then, after the official release, the release branch is merged back into the master and develop branch and tagged.

```
git checkout master  
git merge --no-ff release-3.1.1  
git tag -a 3.1.1
```

```
git checkout develop
git merge --no-ff release-3.1.1
```

Finally, use the following command to delete the release branch after the merge is complete.

```
git branch -d release-3.1.1
```

Hotfix Branch

First, create the hotfix branch using the following command.

```
git checkout -b hotfix-3.1.1.1 master
```

Then, after the official release, merge the hotfix branch back into the master and develop the branch and tag it.

```
git checkout master
git merge --no-ff hotfix-3.1.1.1
git tag -a 3.1.1.1
git checkout develop
git merge --no-ff hotfix-3.1.1.1
```

Finally, use the following command to delete the hotfix branch after the merge is complete.

```
git branch -d hotfix-3.1.1.1
```

Gitflow has several advantages, such as clear branch naming semantics, support for staging environments, and support for multi-version production environments. These advantages have influenced the various Git branch management workflows.

GitHub Flow

GitHub flow is widely used as a lightweight branch management strategy, and some of its key features are listed below.

- The master branch is publishable at any time.
- New branches are pulled from master branches during development, but no distinction is made between feature and hotfix branches.
- Push the new branch to the remote after development is complete.
- Notify the rest of the team for review and discussion by submitting a Pull Request (PR) to the master branch.
- After the PR is accepted, the code is merged into the master branch and triggers continuous deployment.

GitHub Flow is simple enough and thus popular with the open-source community. But sometimes merging code into a master branch doesn't mean that it can be released immediately. For example, many companies have a fixed go-live window where they can only release new versions at certain times, which can cause online versions to fall far behind the master branch.

In addition, the GitLab team came up with GitLab Workflow (GitLab Flow) in 2014. The biggest difference between it and GitHub workflows is the inclusion of environment branches, such as release and production branches.

One Flow

Adam Ruka, in his 2015 article *GitFlow considered harmful*, proposed a new branch management strategy called One Flow. One Flow can be seen as a replacement for Git Workflow. Instead of maintaining two long-term branches, One Flow uses a master branch instead of a develop branch in Gitflow. Thus, both the feature and release branches are pulled from the master branch, while the hotfix branch is pulled from the release branch. Compared to Gitflow, the code in One Flow is cleaner, easier to read, and less expensive to maintain. In addition, Alibaba has proposed a similar branch management strategy, the Aone Flow, with the same core idea of merging the master and develop branches into a single branch.

In actual project development, our team uses multiple branch management strategies at the same time: Gitflow to publish client-facing applications, which can meet the needs of multi-environments and multi-version management while ensuring the quality of applications; One Flow to publish microservice applications, and GitHub flow to manage internal tools. Readers can choose the proper branch management strategy based on the teams' requirements.

3.3.2 *Code Inspection Based on Sonar*

I have heard developers talk about code quality like this: "It works!" "No complaints from customers!" Or "Automated tests are enough!" But is that true? The answer is obviously no.

Code quality should not only rely on automated tests such as unit tests and regression tests, but code inspection is also very important and it can help us find bugs within the code as early as possible. Generally speaking, code inspection is divided into auto inspection and manual review. A common practice is to combine them. For the former, our team has chosen the code inspection tool Sonar. In this section, we will introduce how to use Sonar for code inspection.

Sonar, an open-source code quality management tool, helps us to continuously check code quality. It supports more than 20 major development languages, and developers can either use the built-in rules for checking or easily extend the rules.

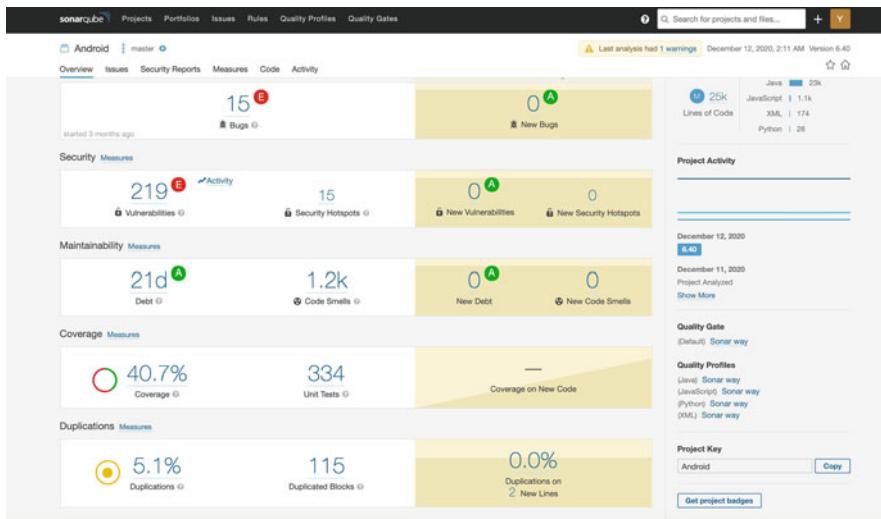


Fig. 3.7 Code quality report

Code Quality Report

Sonar provides real-time code quality reports covering metrics such as reliability, security, maintainability, test coverage, and duplications, as well as Quality Profiles and Quality Gates to measure code quality.

- Reliability: Sonar can check for potential bugs that, if not fixed in time, can affect the function of the software product.
- Security: Sonar has a series of built-in security rules, such as code injection, SQL injection, cookie rule, etc. These rules are mostly proposed by OWASP Top 10, SANS Top 2,5, and CWE. Security issues should be fixed as soon as possible.
- Maintainability: Sonar summarizes common code smells for various languages. Code smells often reveal deep-seated problems that, if left uncorrected, can lead to bad consequences in the future.
- Test coverage: Sonar can collect code coverage reports for unit tests or regression tests.
- Duplications: Duplicate code is almost always the most common code smell. Eliminating duplicate code should be one of the main goals of refactoring.
- Quality Profile: Quality Profile is a set of rules. We can use Sonar default rules collection or customize the rule set. The Quality Profile should be set according to the actual situation so that the whole team follows the same set of standards.
- Quality Gate: Quality Gates define the criteria for judging the quality of code. Take Sonar's default Quality Gate as an example; only when the test coverage is greater than 80%, the duplication is less than 3%, and the reliability, security, and maintainability ratings are not lower than A, the code quality is passed; otherwise, it is marked as failed.

Figure 3.7 shows the code quality report for an application called “Android.” The left side shows the statistics of the code metrics, and the right side shows the quality configuration and quality gates being used by the application.

Continuous Code Inspection

Developers can use the static code-checking plugin SonarLint in their local IDE for local analysis. In addition, Sonar also provides a continuous code inspection process, which combines continuous integration and code quality reporting.

When a developer creates a new PR using Git, it triggers the Jenkins pipeline. We can add analysis steps to the pipeline and use the Sonar Scanner for code analysis, integrated as follows.

```
stage('SonarQube analysis') {
    def scannerHome = tool 'SonarScanner 4.0';
    withSonarQubeEnv {
        sh "${scannerHome}/bin/sonar-scanner"
    }
}
```

The Sonar Analyzer sends quality reports to the Sonar server with the following example analysis parameters.

```
sonar-scanner \
-Dsonar.sources=. \
-Dsonar.host.url=https://sonar.dev.net \
-Dsonar.projectKey=Android
```

The sonar.sources refer to the source code path, sonar.host.url represents the address of the server, and sonar.projectKey is the unique identifier of the application where the source code is located on the Sonar server, and each application should have its projectKey.

The sonar server compares the current analysis results with the latest analysis results of the master branch and generates a PR-based report. We can view the code quality result in the pipeline and on the Sonar web site for detailed quality reports. Figure 3.8 shows an example of the PR quality report. The code quality is marked as Failed because there are two code smells in the latest code, which do not meet the zero code smell requirement of the quality gate.

In addition, Sonar also supports integration with Git and displaying inspection results in PR. To use it, you need to create a GitHub App and make it point to the Sonar server, then configure the GitHub address and App ID in the global configuration of the Sonar server, and finally adjust the parameters of the parser. An example of the analysis in PR is shown below.

```
sonar-scanner \
-Dsonar.pullrequest.key=12345 \
```

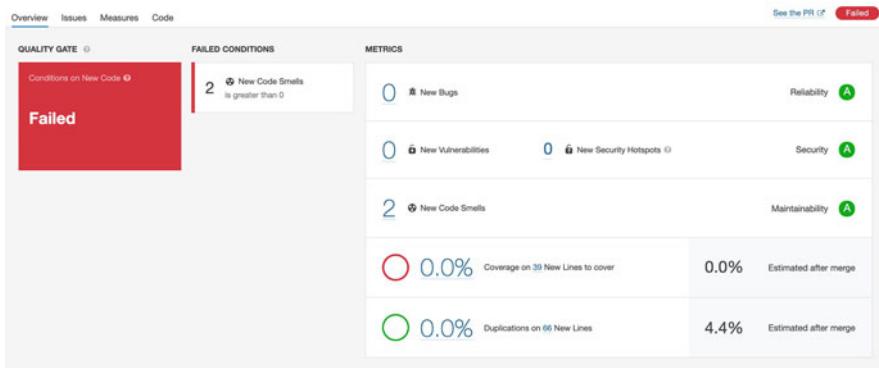


Fig. 3.8 PR quality report in Sonar

```
-Dsonar.pullrequest.branch=my_branch \
-Dsonar.pullrequest.base=master
```

The sonar.pullrequest.key is the ID of the PR, the sonar.pullrequest.branch is the source branch of the PR, and the sonar.pullrequest.base represents the branch to which the PR points.

After the Sonar server generates the code quality report, it writes the results back to the PR. Figure 3.9 shows how the above results are presented in the PR. Code merging should occur after the code has been marked as passed.

Each team has its own set of code specification guidelines or its code quality standards. By using Sonar for continuous inspection, we can standardize the development practices and improve code quality.

3.3.3 *Code Review*

Code Review generally refers to manual review. Code review can identify bugs in the code, unify the coding style, improve engineering ability, and promote the sharing of design and implementation.

Participants in Code Review

Code review participants are divided into two categories: code submitters and code reviewers, and their responsibilities of them are different.

Code submitters should follow the following code of conduct.

- Avoid submitting large PRs. Large PRs, such as those over 1000 lines, can both introduce more bugs and affect review efficiency.

The screenshot shows a GitHub pull request (PR) quality report. At the top, there are navigation links: Conversation (5), Commits (5), Checks (1), and Files changed (5). Below these, a red circular icon with a white 'X' and the text 'fix comment. 96188a4' is displayed. The main content area has a sidebar on the left with sections for SonarQube PR Checks, SonarQube Code Analysis (which is selected and highlighted in blue), and Re-run. The main panel displays the SonarQube analysis results for 'SonarQubePRChecks / SonarQube Code Analysis', which failed 26 days ago in 12s. A large bold heading 'Quality Gate failed' is shown, followed by a 'Failed' button. Below this, it says '2 New Code Smells (is greater than 0)' and provides a link to 'See analysis details on SonarQube'. An 'Additional information' section follows, stating that metrics like bugs, vulnerabilities, and security hotspots might not affect the Quality Gate status but improving them will improve project code quality. It lists '2 Issues': 0 Bugs, 0 Vulnerabilities (and 0 Security Hotspots to review), and 2 Code Smells.

Fig. 3.9 PR quality report in Github

- Write a clear title and clean description of the PR, indicating what the PR is doing and why.
- Try to complete the development of the PR; otherwise, add WIP to the title of the PR to indicate that it is under development.
- Ensure the quality of the code.
- Ensure single PR responsibility and reduce the cost of code reviews.

Code reviewers should follow the following code of conduct.

- Review newly submitted or updated PRs in time.
- Complain less and understand that there is never a perfect code.
- Show respect and patience. Target the code, not the code submitter.
- Provide code improvement suggestions.
- Realize that refactoring of code can happen in future iterations if not necessary.
- Avoid reviewing too much code at once.
- Control the review pace, such as reviewing no less than 500 lines of code per hour.

The Form of Code Review

Code reviews are generally divided into two types: online code reviews and offline code reviews. Online code review is an asynchronous way of code review. The code

committer submits the PR to the code reviewer and then can start the next task. The code reviewers can review the code based on their schedule. In contrast, an offline code review is where the developer explains the code to the team in a conference room. Offline code reviews help reduce communications between code committers and code reviewers and also help the team understand the goals and implications of the code review. We can use a combination of online and offline code reviews.

What to Focus on in the Code Review

The code review process usually focuses on the following issues.

- The code should have good design ideas.
- The code should meet the functional requirements. If it is a front-end change, a UI screenshot is required.
- The implementation of the code should not be too complex and should avoid over-design. Simple and easy-to-read code can reduce maintenance costs.
- The code should have test cases. Test cases should also have good design ideas.
- The code should have good naming rules.
- The code should be documented and commented on. Documentation, such as godoc, is used to explain what the code is used to do. Comments, on the other hand, focus on explaining what and why.
- The code should follow the team's code style guide.

3.3.4 Commit and Merge

Typically, a PR contains one or more commits. Each commit should reflect a small change and have a clear description of that change. We can use the Git command to commit code.

```
git commit -m "my commit message" // Describe the content of the commit
```

The description of the commit is very important. It helps the rest of the team understand what the code means. We may even search through the code commit logs years after the code was committed to figuring out why it was implemented that way. Without an accurate description, none of this would be possible.

Once a PR is finished, it's ready to be committed. When committing, push the local feature branch to the remote, then create a new PR that points to the develop branch or release branch as required. One or more code reviewers can be set up for the PR commit. Generally, you can consider merging once the code review is complete. When merging, the following points should be ensured.

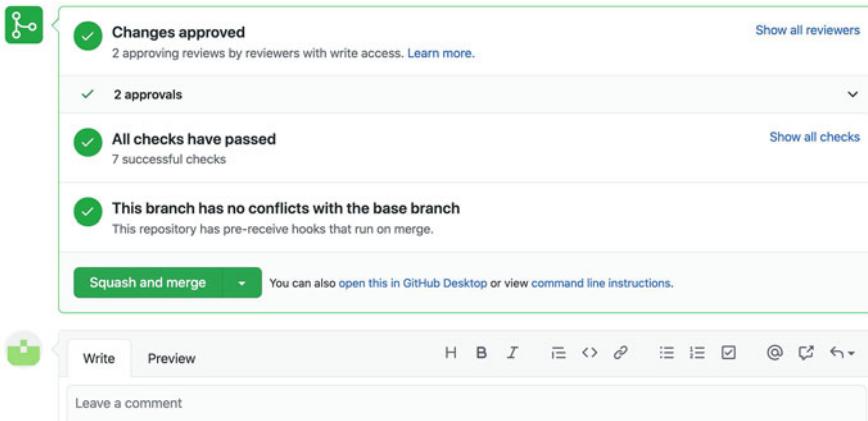


Fig. 3.10 PR example

- The code can only be merged if all reviewers on the PR agree.
- Each PR change triggers continuous integration, such as unit tests and Sonar code inspection, and only when all pass can the code be merged.
- PR merges are performed by one of the code reviewers. No one can merge his or her code.

Figure 3.10 shows an example of a PR merge. As can be seen, the PR was approved by a total of two code reviewers, passed seven checks, and can be merged into the target branch.

In addition, project management tools should be used to track the lifecycle of a PR. Our team uses Jira to track PRs, and each PR, even each commit, is assigned a Jira task number so that the PR is recorded from the time it is created to the time it is closed, and every time it is updated. Team members can use the task number to find the corresponding code change, or they can trace a line of code back to the corresponding task number to understand the code.

3.4 Low-Code Development Platform

With the rise of microservices architecture, the way services are built has evolved again. Building a service quickly is becoming a new challenge. For this reason, our team chose a low-code development platform. In this section, we'll explore what features the low-code development platform has and how to use it to achieve rapid development.

3.4.1 Low Code and Development Platform

Low code is a new development paradigm that allows developers to quickly build services with little or no code. The Low-Code Development Platform (LCDP) is a development platform based on low code and visualization. It was first defined by Forrester Research in June 2014.

Platforms that enable rapid application delivery with a minimum of hand-coding, and quick setup and deployment, for systems of engagement.

It is easy to tell from the definition that a low-code development platform is an important tool to improve efficiency and reduce costs. Thus, a low-code development platform should have the following three capabilities.

- **Visual Programming**

The Low-Code Development Platform can be thought of as an IDE that allows users to create services from its library of components in a visual or even drag-and-drop manner, like building blocks. Compared to the visualization capabilities supported by traditional IDEs such as Visual Studio's MFC, the Low-Code Development Platform supports end-to-end visual programming.

- **Full Lifecycle Management**

The Low-Code Development Platform supports the full lifecycle management of services. Through the platform, we can not only design and develop services easily, but also deploy them with one click, and meet the observability requirements of services. The platform's management of the service lifecycle also brings aggregate effect, making the platform an ecosystem of services.

- **Extensibility**

The platform helps users to build different services by building different templates. The service templates can be either service code frameworks based on different languages or service frameworks combined with different cloud services.

3.4.2 Low-Code Development Platform Practices

In cloud-native architecture, a low-code development platform is also important. It helps us to build a service quickly so that we can focus more on business innovation, thus greatly reducing the business iteration cycle. Our team has built a low-code development platform, internally called Bingo, which provides a set of visual interfaces to support service template management and full lifecycle management of services.

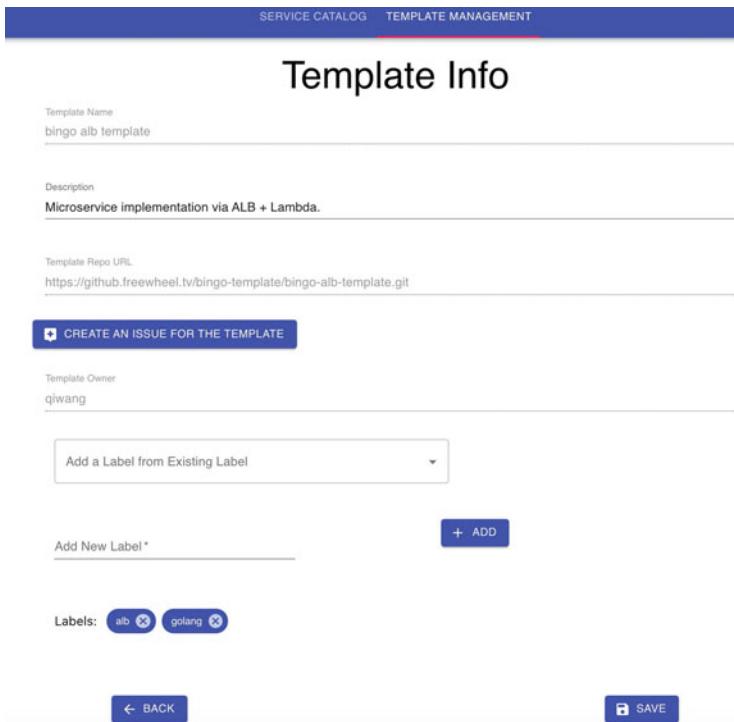


Fig. 3.11 Template information

Service Template Management

We provide a template for each type of service, such as the Asynchronous Job Template, Cron Job Template, Microservice Template, Serverless Template, etc. Each template defines a best practice for a specific use case. When team members create a new service using Bingo, they can simply choose the appropriate template based on the business scenario. Team members can also add new templates if needed.

Each template has a detail page with information about the contributor, name, Git repository of the template code, detailed description of the scenario, tags, etc. For example, Fig. 3.11 shows a template that combines the AWS Application Load Balancer (ALB) and AWS Lambda build APIs, with the template's Repo URL containing the Git URL where the template code is located. Team members can click the “Template Feedback” button to provide feedback on a template.

The template code is the key to the template. The template code consists of instructions, a hello world, a Makefile file, a configuration file, a deployment description file, a pipeline file, etc. Figure 3.12 shows the directory structure of the template code of ALB + Lambda.

config	update config of stg	4 months ago
functions/hello	add comment	4 months ago
master_pipeline	update mp	6 months ago
.gitignore	FW-29679 ut and readme	8 months ago
Makefile	Update Makefile	2 months ago
README.md	FW-29679 ut and readme	8 months ago
bingo.yml	Update bingo.yml	5 months ago
go.mod	modify bingo-utils version	7 months ago
go.sum	modify bingo-utils version	7 months ago

Fig. 3.12 Template code

- The instruction manual, README.md, contains the name of the template and instructions for its use.
- hello world is a piece of executable code, such as an API that returns hello world, as shown in the following example.

```
func GetHelloWorld(ctx context.Context, req *model.Request) (interface{}, error) {
    InitConfig()

    log.Println("Hello World!")
    log.Printf("Request path: %v", req.Path)

    // Get the parameters from the Lambda environment variable
    DomainServiceUrl := os.Getenv("DomainService")

    message := fmt.Sprintf("Message: Hello, Bingo! Domain service url is [%s] (Time: %s) ", DomainServiceUrl, time.Now().UTC().String())
    return message, nil
}

// Initialize the Viper configuration object based on environment variables
func InitConfig() {
    v := viper.New()
    v.AutomaticEnv()
    config.Set(&config.Config{Viper: *v})
}
```

- The Makefile file defines commands for unit testing, test coverage, packaging, etc. These commands will be integrated into the continuous integration pipeline.
- Configuration files are configuration variables that are given to the development environment, staging environment, production environment, and other environments.
- The deployment description file is a custom YAML file that uses to describe the deployment content of the cloud-native services. When the service is deployed, the YAML file is converted into a .tf file that is recognized by Terraform, an infrastructure automation tool. An example of a deployment description file is shown below.

```
# bingo-alb-template
application: bingo-alb-template
```

```

common:
tags:
  Project: Bingo
  App: ALBTemplate

functions:
- name: HelloWorld # required!
  handler: hello # required! binary name
  runtime: go1.x # optional, default go1.x
  description: API of bingo hello # optional
  timeout: 10 #optional, default 6s
  environment:
    - hello_env

events:
alb:
- priority: 1
  conditions:
    path: # path is the array to support multiple paths
      - /bingo/hello
    method: # method is an array to support multiple HTTP METHOD
      - GET
  functions:
    - name: HelloWorld
      weight: 100

```

- Jenkins files are template files used to generate continuous integration and continuous deployment pipelines.

Full Lifecycle Management

The bingo platform supports the full lifecycle management of services. The full lifecycle is shown in Fig. 3.13, which refers to the process from design to development to integration, deployment, and maintenance, in each of which the platform provides corresponding support.

During the design phase, the platform provides service design best practices through service templates. Team members can choose the proper template without designing the service from scratch.

During the development phase, the platform supports quick building based on service templates. To create a new service, you need to select a template and fill in the service name, description, Git repository name, Git organization name,



Fig. 3.13 The full lifecycle

The screenshot shows a web-based service creation interface. At the top, there are two tabs: "SERVICE CATALOG" and "TEMPLATE MANAGEMENT". Below the tabs, the main title is "Create A New Service".
The first section contains a dropdown menu labeled "Skip Creating Repo" with options "no" and "yes".
The second section is a dropdown menu labeled "Select A Template".
The third section contains three input fields: "App Name *", "App Repo Name *", and "Repo Org Name".
The fourth section is a text input field labeled "Description *".
The fifth section is a text input field labeled "CI Pipeline *" containing the URL https://jenkins.dev.fwci.aws.fwmrm.net/job/UI/job/UI_CI/job/bingo/job/Bingo_FullCI_Pipeline/.
The sixth section contains three dropdown menus labeled "Project Tag", "App Tag", and "Service Tag".
At the bottom left is a blue "CANCEL" button with a left arrow icon. At the bottom right is a blue "CREATE" button with a checkmark icon.

Fig. 3.14 Service creation

continuous integration pipeline, and various tags, as shown in Fig. 3.14. Among them, the Git repository and Git organization are used to specifying the location of the service code.

After filling in the service information on Bingo UI, click the “CREATE” button at the bottom right corner to automatically create a service; the process is as follows.

1. Verify that the Git organization for the service exists, and exit if it does not.
2. Verify that the Git repository for the service exists, and exit if it does; otherwise, create a Git repository for the service.
3. Give the current user developer permissions for the Git repository.
4. Find the Git repository of the corresponding template according to the name of the service template, and then clone it to the server.
5. Edit the template code according to user requirements, such as replacing the template name with the service name, adding or reducing the common library, etc.
6. Modify the remote code location from the template’s Git repository to the service’s Git repository.

7. Use the Git command to commit the code and push it to the remote end to complete the generation of the scaffold code.
8. Clean up the temporary files on the server side and write the data to the database.
9. Developers write business code based on the scaffold code in the Git repository.

The Bingo platform also supports continuous integration and continuous deployment. Developers committing new code triggers a continuous integration pipeline. The pipeline includes unit tests, regression tests, and test coverage reports. The pipeline also packages the service code into two tar packages for deployment. The two packages contain the program binary and the deployment description file. After developers finish development, they can complete one-click deployment on the platform. Taking the deployment of Serverless services as an example, one-click deployment triggers the continuous deployment pipeline, which first converts the deployment description file into a .tf file that Terraform can recognize, then uploads the service binaries to AWS S3, and finally completes the service deployment using Terraform.

The Bingo platform also supports the operation and maintenance of the service. The Bingo platform introduces the ELK-based logging solution and the Jaeger-based distributed tracking system.

3.5 Service Operation and Maintenance Platform

With the increase in the number of services, we encountered big challenges related to the operation and maintenance of the services. To solve this problem, we have built a service operation and maintenance platform.

3.5.1 *Problems to Be Solved by the Platform*

We often face the following problems.

- How to perform simple debugging and quickly locate the problem when the application goes wrong?
- Distributed systems tend to introduce data inconsistencies; how can such data be monitored?
- In an asynchronous message-based system, did the business function not complete properly, did the producer not send the message out, or did the consumer not receive the message?
- Why is the updated data in the database delayed and when does the cached data expire?
- What background tasks are being executed, how are they scheduled, and what are the reasons for execution failure?

Previously it was difficult to find a unified solution to the above business problems, and sometimes we even had to rely on monitoring and debugging in the production environment. However, during the evolution of microservice architecture, we found that although the services are different, they share similar development practices and even the same framework, which allows us to maintain the services in a unified way. Therefore, our team decided to build a service operation and maintenance platform from scratch to solve the pain points of service governance.

3.5.2 *Platform Architecture*

When designing the service operation and maintenance platform, we have the following considerations.

- The platform should be tied to the team's business.
- When new requirements arise, they can be quickly scaled horizontally.
- Avoid overlap with the team's monitoring platform and other existing products.

Based on the above requirements, our team built a service operation and maintenance platform, an internal project called Falcon, which consists of four parts.

- Falcon front-end provides all web resources. The development and deployment of the front-end are independent of the back-end. The front-end uses React.
- The Falcon back-end, the platform's back-end server, is responsible for the platform's business logic. The back-end needs to interact with the microservices and public components in the cluster. The back-end uses Node.js and a data interface in JSON format.
- MySQL as a database, such as information of logged-in users, collected business data, etc.
- Redis to implement features such as timed tasks.

We packaged the Falcon front-end, Falcon back-end, Redis, and database separately and deployed them as Kubernetes Pods so that they were in the same cluster as the other services, as shown in Fig. 3.15.

Requests are first forwarded through the load balancer to the Ingress Gateway and finally to the corresponding Pod of Falcon, which interacts with the cluster, such as listening to messages delivered by Kafka, reading data from the cluster Redis, and calling the microservice interface.

3.5.3 *Platform Modules*

Falcon platform contains several modules: user management module, data monitoring module, background job module, asynchronous message module, business cache

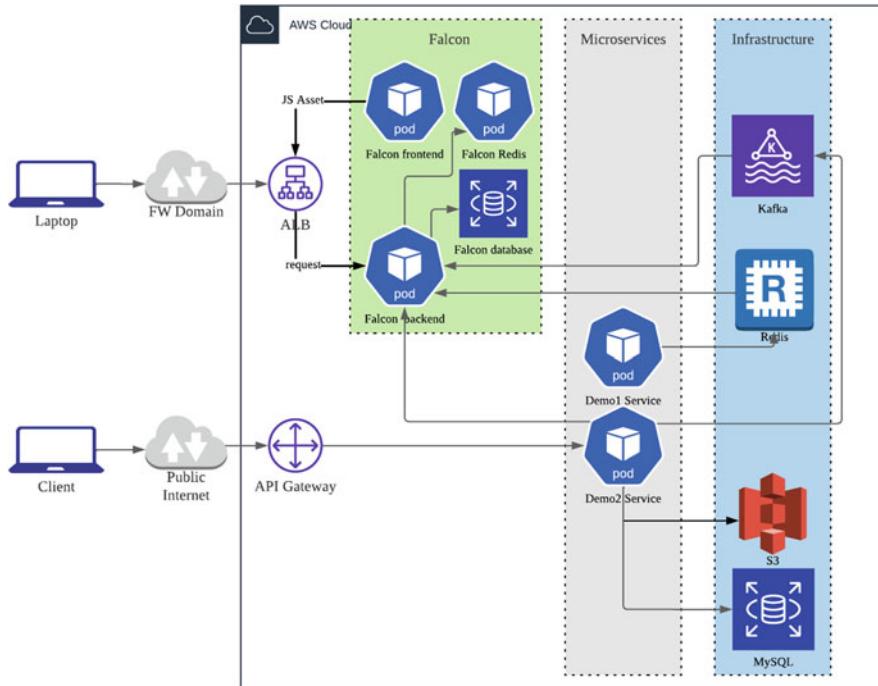


Fig. 3.15 Falcon architecture

module, online debugging module, and change history module. The Falcon navigation home page is shown in Fig. 3.16; some of the modules are still under exploration and development.

Users and Permissions

Company employees use Light Directory Access Protocol (LDAP) for unified login authentication, so Falcon's back-end also integrates LDAP to authenticate logged-in users. When a user logs in for the first time, Falcon synchronizes the user's information in the database so that permissions can be configured later. User permissions are managed by module, and permissions between modules are independent of each other.

Data Monitors

The Data Monitoring module is designed to monitor abnormal business data. After migrating from a monolithic application to a microservice architecture, many data insertions, deletions, and updates have changed from a single database transaction to

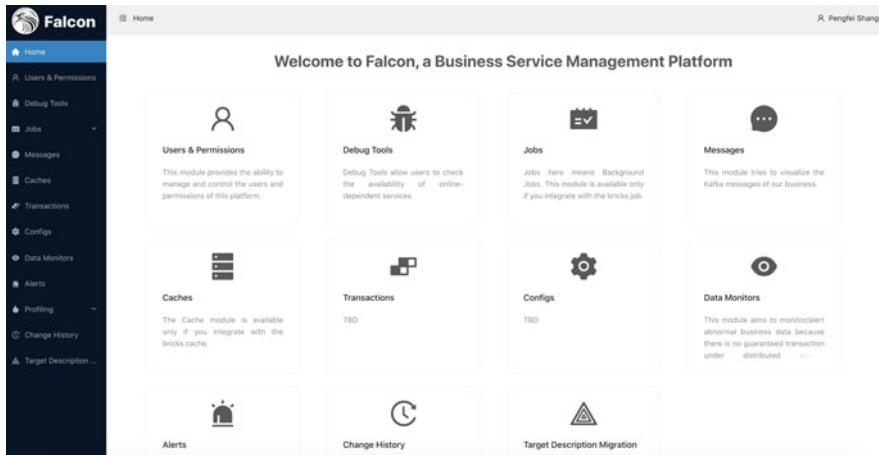


Fig. 3.16 Home page

a distributed transaction involving multiple microservices or even multiple databases. Falcon provides an entry point for monitoring and alerting on dirty data, and users can monitor specific data by providing a SQL statement or an interface call.

After the Falcon back-end is started, it will load the monitoring configuration from the database and initialize a scheduled task to add it to the queue, execute SQL or call the interface at regular intervals, and write the execution result back to the database. The Falcon back-end compares the execution results with the user's pre-set parameters and triggers alerts and notifications to subscribers if dirty data is found. Subscribers can change monitoring settings in real time, and the Falcon back-end will persist the user's changes and update the task queue in real-time.

Jobs

Our team's back-end job framework supports creating, executing, retrying, and destroying back-end jobs but does not support visual job management. To address this pain point, we added a background job visualization module to Falcon. Users can see which jobs are executing, which jobs are queued in the queue, which jobs have failed, which jobs have been retried, what time the timed jobs are scheduled, etc. In addition, the management page also displays detailed information about the parameters and error contents of failed jobs and provides a manual retry function.

Messages

As clusters become larger and larger, asynchronous messaging mechanisms are being used more and more widely. Our team uses Kafka to implement asynchronous

message processing. In the past, we could only know the status of messages by querying logs, which was very unfriendly. In the asynchronous messaging module, we visualize Kafka business messages by creating new consumers to listen to messages.

Caches

To improve the processing and responsiveness of the application and reduce the pressure on the database from the business layer, we introduced a caching layer to put the commonly used and unchangeable data into the cache. Similar to the problems handled by the background job module and asynchronous messaging module, we often have no way to know what is stored in the cache and when it was updated. A typical scenario is that the data in the database is updated, but we do not know when the data took effect, which leads to errors in locating the problem. In the business cache module, we visualize the cached data and provide a search function. Users can see what data is cached, what the amount of data is, when the data expires, etc.

Debug Tools

Applications often rely on third-party services, and in a production environment, when problems occur with these third-party services, it is difficult for us to quickly locate the problem. For example, when the application is slow to respond, it is difficult for us to determine whether it is slow to query the database or slow to call the downstream services. Different services can implement different debugging interfaces and provide debugging information according to different business situations. The online debugging module integrates these debugging interfaces. Users can manually trigger the interfaces in the platform to see the execution of the whole chain. The online debugging module can help engineers quickly locate the cause of errors and save time.

Change History

The operations that occur on the platform are recorded using the change history module to facilitate better tracking. Considering that the platform itself does not have particularly complex operations, and at the same time updates will not be particularly frequent, we keep the full content of the usage log instead of the variable content, i.e., when an object changes, we take a full backup of the snapshot of the original object. With this feature, we can see the state of the object at a given moment, and we can also easily see which fields have changed.

3.6 Service Middle Platform

In the past two years, the middle platform has become a hot topic, and related articles have been published by various technology communities and media outlets. But different people have different understandings of what a middle platform is. If the middle platform is a solution, what problem is it used to solve? And how is a microservice system's middle platform built? This section will try to answer the above questions and introduce our team's thinking and practice about the middle platform.

3.6.1 What Is a Middle Platform

At the end of 2015, Alibaba Group launched a comprehensive middle platform strategy to build a new “large middle platform and small front-end” organization and business mechanism. The concept of the middle platform was born and soon became a shining star in the software industry. I even heard that some companies started the middle platform strategy and directly renamed the technical back-end platform to a business middle platform. Many companies have indeed been making efforts for many years on the “front-end + back-end” architecture, but this does not mean that the back-end platform is equivalent to the middle platform. For this reason, let's first define “front-end + back-end.”

- The front-end platform is the system used by the enterprise to deliver to the end user (customer) and is the platform for the enterprise to interact with the customer; for example, the website and app directly accessed by the user can be counted as the front-end.
- The back-end platform is the back-end system, computing platform, and infrastructure that manages the core information resources (data) of the enterprise. The back-end does not interact directly with end users and does not (and should not) have business attributes.

Stability and reliability are the goals pursued by the back-end. The front-end, on the other hand, needs to respond quickly to frequent changes in customer demand because it has to interact with customers. Therefore, there are irreconcilable conflicts between the front-end and back-end in terms of target requirements and response speed. They are like two gears, one large and one small, which are difficult to run smoothly and harmoniously because of the difference in tooth density ratio.

The birth of the middle platform breaks the traditional “front-end + back-end” architecture model. As shown in Fig. 3.17, the middle platform is like a gear between the front-end platform and the back-end platform, matching the rotation rate of the front-end and the back-end to find a dynamic balance between stability and flexibility. We have reasons to believe that the middle platform is made for the front-end. So, how to understand the middle platform?

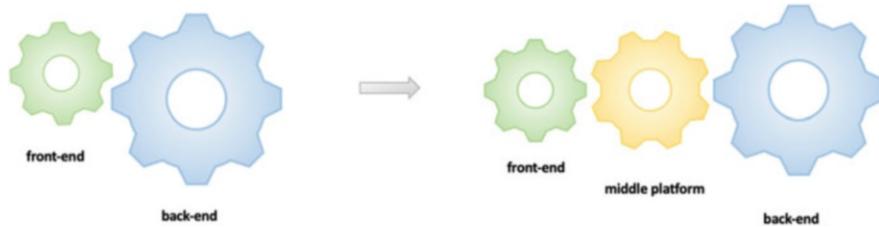


Fig. 3.17 Frontend and backend

There are a thousand Hamlets in a thousand people's eyes, and different people have different interpretations of the middle platform.

- Some people think that a middle platform is a business platform, and public business systems like inventory, logistics, orders, and payments of e-commerce companies can be called business middle platforms. Similarly, a microservice business system can also be called a business middle platform.
- Some people think that the middle platform is the data platform, and platforms like data collection, data processing, and data analysis can be called the data middle platform.
- Some people think that the middle platform is the technology platform, like a microservice development framework, PaaS platform, etc., which can be called the technology middle platform.
- Some people believe that the middle platform is the organizational structure platform and that the middle platform cannot be built without a shared functional team.
- Some people think that the middle platform is the product of the natural evolution of the platform. When the business of an enterprise grows faster than the service capacity load of the platform, the business with commonality needs to be abstracted and precipitated to quickly support new business development.

In the author's opinion, each of the above interpretations has its validity. There is no need for us to give a unified definition of the middle platform; instead, it is more important to understand the essence of the middle platform.

We can use three keywords to describe the middle platform: sharing, integration, and innovation. By sharing, we mean that the middle platform should abstract the business with commonality, provide reuse, and examine the whole business from the perspective of the enterprise. By integration, it means that the middle platform should seamlessly connect the front-end and back-end and eliminate the contradiction between the front-end and back-end. By innovation, it means that the middle platform should innovate for the front-end services. The form of the middle platform is not important; instead, empowering the business by building the middle platform is important.

As our team migrated to a microservices architecture, we gradually realized that businesses with commonalities should be abstracted out. In particular, we found that

these common services could effectively help us build new product lines. Building a middle platform became a matter of course. Next, I will introduce the way of building the middle platform with the team's practice.

3.6.2 *The Road to Building a Middle Platform*

No system is built overnight, and this is especially true for the middle platform. After more than a year of development, the team has built a middle platform architecture suitable for its own business through continuous polishing, trial, and refactoring. The development process can be divided into three phases, as shown in Fig. 3.18.

Collect Requirements and Build Teams

When we migrated our business systems from a monolithic structure to a microservice architecture a few years ago, we did so through a bottom-up approach, splitting services based on business capabilities. The biggest advantage of this approach was that it allowed for a fast build and development process and early completion of the migration. However, its disadvantage is also obvious: there is no unified planning and design, and the whole system lacks a common framework and service governance capability. To address this issue, we proposed the Business Service Architecture and Practice (BSAP) project, which aims to improve and optimize the existing microservice architecture, provide reusable service governance capabilities for each business line, and provide a set of public libraries and services. The BSAP project aims to improve and optimize the existing microservice architecture, provide reusable service governance capabilities for each business line, and provide a set of public libraries and middleware to improve the efficiency of microservice development. This is where our middle platform was born.

Unlike Alibaba's approach of establishing a middle platform strategy first and then unifying development, we didn't want to build a middle platform at the beginning of the project. Our initial intention was simple: we wanted to abstract similar business logic as components or class libraries for reuse. As the number of middleware and class libraries grew, we realized that the collection of components we were building was essentially a business middle platform.

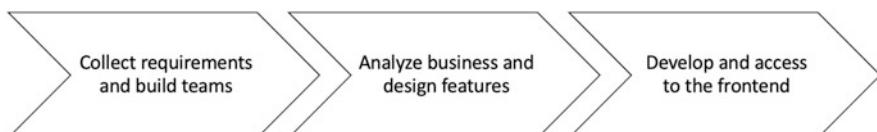


Fig. 3.18 The three phases of building middle platform

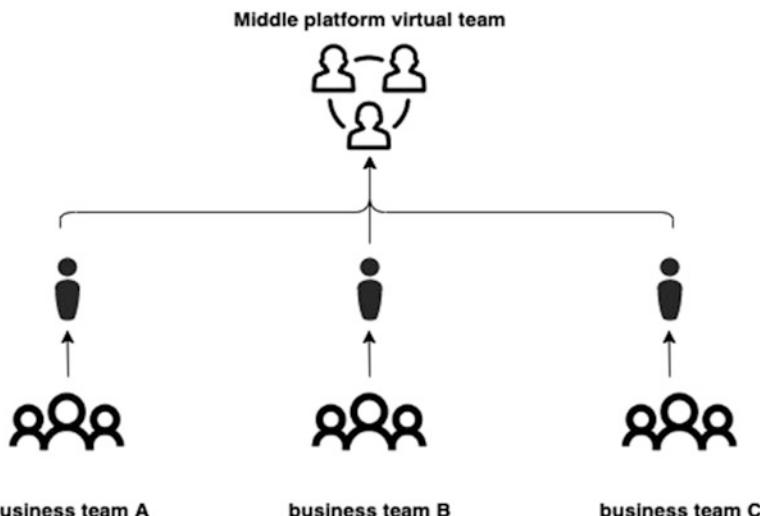


Fig. 3.19 Virtual team

From the perspective of investment structure, our middle platform team is formed through the “crowdfunding model”; the core developers of each business line participate in the project, describe their business needs and pain points, and propose solutions. These contents will be analyzed and discussed in the BSAP meeting; if it is considered valuable, the topic will formally enter the development phase. The development team is the one who proposes the requirements, and they have a deep understanding of the pain points, so they don’t have to worry about decoupling the requirements from the implementation.

Each project team is formed voluntarily and uses its spare time to complete development tasks. Compared with the “investment model,” the crowdfunding model does not require the special deployment of developers to form independent teams, which has greater advantages in terms of human resources costs, management costs, and willingness to develop. The organizational structure of the middle platform team is shown in Fig. 3.19.

The middle platform team is a shared services team that serves the front-end. A large middle platform is difficult to meet the business needs of the front-end in the short term due to its long-term nature and complexity, and the middle platform team is likely to have internal resource competition because it has to serve multiple front-end businesses. Our middle platform is built up in a way similar to a jigsaw puzzle, which can quickly respond to the needs of the front-end. This “small and fast” elite special force is flexible, can change from one project to another, and is the most suitable middle platform team for small and medium-sized enterprises.

Analyze Business and Design Features

Once the requirements are defined, it is time to move on to the design phase. Like any other software development process, design is an indispensable phase, translating business modeling into technical solutions and ensuring correct implementation.

For the middle platform, the design phase has its special feature: by analyzing the business of each domain, we find common capabilities that can be abstracted out. Because the middle platform is to serve multiple front-end business lines, it must analyze the overall business and find common parts to achieve the core goal of reuse. If it only starts from a single business and only meets the current requirements, it is just like implementing its unique business logic for one microservice only. To avoid this situation, during the topic analysis phase, we will brainstorm for a collision of ideas. When a person is describing his or her requirements, people with the same or similar pain points will also resonate and put forward their complementary views, and finally integrate a technical solution that meets both generic and specific requirements.

It is important to note that the so-called common capabilities include business data, business functions, and common technical capabilities. For example, Placement is business data that is used by all business lines, but at the same time, it has certain variants with additional forecasting properties in the ad forecasting business and intermediary-related properties in the partner business. They both share the basic data and at the same time have their special fields. For such cases, we will abstract the operation of the core data model as a template, and each business line will customize it.

There are also many examples of abstraction of generic technical capabilities. For example, to make it easier to develop a new microservice, we designed a lightweight service communication layer framework. The new service only needs to implement the application initialization interface (App Initializer) and define the corresponding port number in the configuration file to implement a web server that supports both HTTP and gRPC protocols, and ServerOption can add various interceptors implemented in the middle platform to complete a series of service governance functions such as tracking, request logging, and API permission control. The developers of the new service only need to define their business interfaces and implement them in the standard Protobuf file.

Overall, the main work of the design phase is to conduct a root cause analysis of the identified pain points, then conduct domain design based on multiple business lines, discuss the overlap of the business and abstract the common business, introduce the middle platform architecture, and establish the corresponding solution; the process is shown in Fig. 3.20.

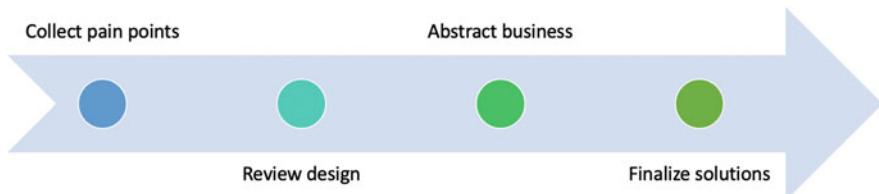


Fig. 3.20 The design process

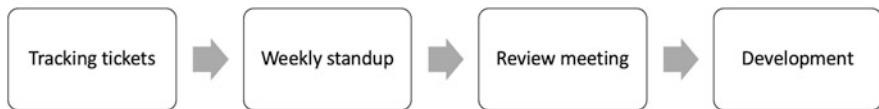


Fig. 3.21 The development process

Develop and Access the Front-End

In the implementation phase, we use the Lean Startup’s Minimum Viable Product (MVP) strategy, which means that the development team provides a minimum viable product, gets feedback from users, and continues to iterate on it rapidly to bring the product to a stable and perfect form.

The MVP strategy works well for startup teams to quickly validate the team’s goals through trial and error, thus positioning the core value attributes of the product. In the process of building the middle platform, each of our crowdfunding squads is a typical startup team, first solving existing pain points through a most simplified implementation solution, and then gradually refining and expanding it to meet the needs of different business lines.

In the development process, we require each task to be tracked by Jira Ticket. In the weekly meeting, each team will update the progress of the development tasks and hold special review meetings in the design, development, and implementation stages to ensure the reliability and controllability of the whole implementation process as much as possible, as shown in Fig. 3.21.

Our middle platform users are microservice developers in various lines of business, and the demand for middle platform capabilities from these developers stems from the customer’s demand for the product. Thus, business requirements drive the needs of the middle platform users (developers), and user requirements drive the middle platform capability. In this demand chain, the business line developers play both Party A and Party B. As seed users, they plug their development results into the business microservices they are responsible for. The service then naturally becomes a pilot of the middle platform. Once the reliability and stability are confirmed, it is then rolled out to other business lines for access.

There are generally two types of middle platform access: self-service access and one-stop access.

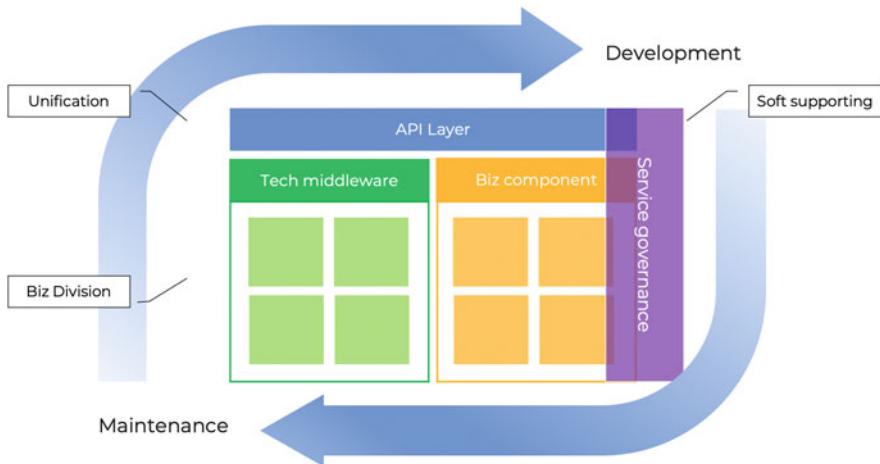


Fig. 3.22 Architecture design

- Self-service access: As the name implies, the accessing party completes the integration with the middle platform components by themselves, and of course, the middle platform developers will provide a series of technical support such as documentation, examples, and training throughout the process.
- One-stop access: The middle platform developers help the user to complete the integration work, including coding, configuration, and so on. This approach is generally used for component upgrades, where code changes are minimal and risk-controlled, and the code holder of the accessing party only needs to review the changes.

In addition, to share the results on a larger scale within the company, we also built a website to provide design documents and user manuals for each component of the business middle platform so that other teams can also access the middle platform in a self-service manner, thus achieving cost reduction and efficiency improvement and technology sharing within the company. After more than a year of effort, our middle platform has become more and more mature. Its architectural design is shown in Fig. 3.22.

The emergence of the middle platform has changed the way business is developed and delivered, and accelerated the iteration and evolution cycle of products. We have reasons to believe that the middle platform will eventually become an important tool to assist the implementation of microservices architecture.

3.7 Summary

The quality of development determines the success or failure of a software product. In this chapter, we discuss the development and operation of microservices. Section 3.1 describes the process of Scrum-based agile development. Section 3.2 introduces different software runtime environments. Section 3.3 covers code management practices. Sections 3.4 and 3.5 introduce the team's self-developed low-code development platform and service operation and maintenance platform; thus, readers will have a clear understanding of how to build and maintain microservices. Finally, Section 3.6 provides insight into the construction of a middle platform in a microservice architecture.

Chapter 4

Microservice Traffic Management



For monolithic applications, there is only inbound and outbound traffic. For microservice architecture, there is another traffic from service-to-service invocation. In a complex network topology consisting of many services, traffic is far more complex than that of a monolithic application. Without policies to manage traffic, the behavior and state of the application will be uncontrollable. Traffic control allows us to gain insight into the system state and implement service governance capabilities, such as service discovery, dynamic routing, traffic shifting, etc.

Service mesh is the preferred solution for traffic management in the cloud-native era. With declarative configuration, applications can control traffic and that is transparent to the application. This means that microservice applications do not need to make any code-level changes. Over the past 2 years, our team has implemented traffic management through the Istio service mesh. According to our practices, this chapter will provide the reader with how to use Istio to control traffic for microservices applications.

4.1 Traffic Management in the Cloud-Native Era

There are many similarities between application traffic management and real-life traffic flow management. Traffic management involves setting up traffic lights to control traffic flow at intersections, adding lanes to improve capacity, adding overpasses to divert traffic, setting up warning signs and roadblocks to guide traffic around, and monitoring and analyzing real-time traffic information by installing cameras and speed measurements. Many similar scenarios can be found in the traffic management of microservice applications. For example, routing requests to specified target addresses (equivalent to road signs), using load balancer, circuit breaker, and rate limiter, setting black/white lists to control access, and analyzing user behavior and system status through traffic metrics, all of these are traffic management tools.

4.1.1 Flow Type

Before introducing the cloud-native traffic management solution, let's understand the two types of traffic: north-south traffic and east-west traffic. For example, suppose a user accesses a UI page through a browser, at which time the browser sends a request to back-end service A; back-end service A wants to complete the user's business operation, it needs to call another back-end service B, and then return the result to the user after the whole process is completed. In this example, the request sent from the user's browser to back-end service A is the north-south traffic, while the request from back-end service A to call back-end service B is the east-west traffic, as shown in Fig. 4.1.

Regarding the content in Fig. 4.1, we can understand it in this way

- North-South traffic: traffic from the client to the server or client-to-server traffic. For traditional data centers, it is the traffic that comes in from outside or leaves the data center; for microservices, it is the traffic that flows from outside the application to inside the application or from inside the application to outside the application.
- East-west traffic: It can be thought of as server-to-server traffic. For a traditional data center, it represents traffic between different servers within the data center or between different data centers. For microservices, it represents the service-to-service traffic in an application.

The main reason for using orientation to describe traffic is that when drawing the network topology, the network components entering the system are drawn at the top (north), the application side is drawn at the bottom (south), and the invocation relationships between the servers, or services within the application, are drawn horizontally (east and west). In a microservice architecture, east-west traffic is

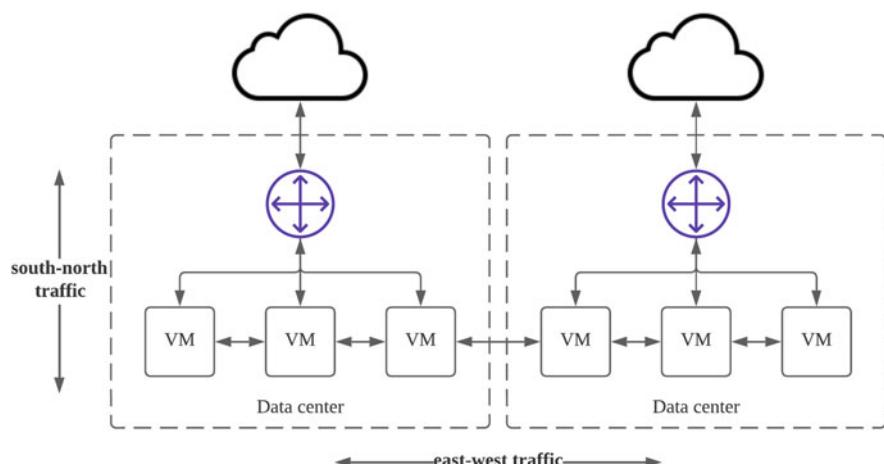


Fig. 4.1 South-north/east-west traffic

much larger than north-south traffic. This is because the same user request is likely to be done collaboratively by several different services, i.e., a north-south flow may generate multiple east-west flows. Therefore, east-west traffic management in microservice applications becomes a top priority, and service mesh, a cloud-native solution for managing east-west traffic, is born.

4.1.2 Service Mesh

As mentioned earlier, service-to-service traffic management is a must-have capability for microservices applications, and how to implement this capability is one of the directions in the industry that has been explored since the birth of microservices architecture. For example, suppose we want to implement a canary release of a microservice application without a service mesh and without any third-party libraries, a straightforward way to implement it is to add a proxy (such as Nginx) between services, and split the traffic by modifying the traffic forwarding weights in the configuration.

```
upstream svc {  
    server svc1.example.com weight=5;  
    server svc2.example.com weight=95;  
}
```

This approach couples traffic management needs with the infrastructure. For applications that contain multiple services, it is difficult to imagine that a proxy needs to be added before each service and the traffic configuration for each service needs to be maintained manually.

Service mesh solves these problems well by allowing you to easily manage traffic flexibly in a cloud-native way (declarative configuration) and without depending on libraries transparent to the application. So what exactly is a service mesh? What can service mesh be used for? We describe it in detail below.

Definition of Service Mesh

The term service mesh is generally considered to have been first raised by William Morgan, CEO of Buoyant, who wrote a blog post in April 2017 called “What’s a service mesh? And why do I need one?” The post was released along with their service mesh product, Linkerd. In the article, the service mesh is defined as follows.

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It’s responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud-native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code without the application needing to be aware.

There are four key points in this definition, through which you can understand exactly what a service mesh is.

- Essence: an infrastructure for service-to-service communication
- Functionality: request transmission
- Product model: a group of proxies
- Features: transparent to the application

The service mesh separates service communication and related management functions from the business logic to the infrastructure layer. In cloud-native applications, with hundreds or more services, the service call chain for a single request can be so long that separate communication processing is necessary; otherwise, you cannot easily manage, monitor, and track the communication across the application. And that's where the service mesh comes in.

What the Service Mesh Can Do

As the infrastructure responsible for service communication in a microservice architecture, service mesh can handle most of the problems in service communication. In simple terms, the service mesh implements traffic management by adding a service proxy (sidecar) in front of each application service and then letting this service proxy hijack traffic and forward it to the application service. Since the service proxy can hold the traffic, it can perform various processes on the traffic as needed, such as distributing the request to different service versions based on the request header. This is a bit like what we know as interceptors and filters. The service mesh is generally considered to have four main functions.

- Traffic management: This is the most core function of the service mesh and the focus of this chapter. Dynamic routing, for example, allows you to dynamically determine which services to request by configuring routing rules. Do requests need to be routed to production or pre-release environments, test or run versions, just for logged-in users or all users? All of these routing rules can be configured in a declarative pattern.
- Policies: Add some request control policies to the application, such as black/white list, rate limit, etc.
- Security: Since the traffic is held, it is natural to do a series of authentication and authorization for the traffic, such as adding TLS for the traffic.
- Observability: Requests necessarily contain a lot of available information, which can be collected through the service mesh and reported to the corresponding back-end processing system (e.g., Prometheus), and then displayed in various visualizations so that developers can fully monitor and observe the operational status of the application.

Advantages of Service Mesh

The biggest advantage of the service mesh is that it is transparent to the application. All traffic management-related functions are implemented by the sidecar proxy, and in the vast majority of cases, the application code does not need to be modified in any way. At the same time, the sidecar proxy is a separate process (usually deployed as a container along with application service, sharing the same network environment as the application's container), and there is no need to use dependency packages. In addition, the sidecar proxy forwards requests by automatically hijacking traffic, and there is no need to add any configuration to the application service. Thus, being insensitive to the application is the biggest advantage over traditional solutions such as public frameworks. It is also because of the transparency that service mesh is suitable for more general and broad scenarios.

- Decoupled from the application: Nonfunctional requirements are implemented by the sidecar proxy; no need to add control logic to the business logic.
- Cloud-native: The service mesh has the typical characteristics of cloud-native technologies and is used through declarative configuration, which is nonintrusive and easy to maintain for applications.
- Multi-language support: As a transparent proxy running independently of the technical stack, it supports multi-language heterogeneous microservice applications.
- Multi-protocol support: Multiple protocols can be supported.

In addition, sidecar proxies can generally extend their own capabilities through APIs, and with the addition of WebAssembly technology, the functions of the service mesh can be easily extended. With the gradual maturity of SMI and UDPA standards, flexible replacement of control plane and data plane will also become possible.

4.2 Service Discovery

In Chap. 2, we introduced the basic concepts and two patterns of service discovery, and understood the importance of service discovery for microservice architecture. In this section, we will focus on how to choose a reasonable service discovery mechanism to complete service-to-service communication after the application is deployed on the cloud.

4.2.1 Traditional Services Discovery Problems on the Cloud

Traditional service discovery solutions generally have a registry. We take Eureka of Spring Cloud framework as an example. Eureka contains two parts: server-side and client-side. The server-side is a registry, providing service registration and discovery capabilities; client-side registers their address information to the Eureka server and periodically sends heartbeats to update the information. The service consumer relies on the client to also obtain registration information from the server and cache it locally, and update the service state by periodically refreshing it. The relationship between the three is shown in Fig. 4.2.

Synchronization of Eureka is done by heartbeat. The client sends a heartbeat every 30 s by default, and the status of the client service is reported through the heartbeat. By default, if the Eureka server does not receive a renewal from the client within 90 s, the client instance will be removed from the registry. As you can know, Eureka state synchronization is done asynchronously by heartbeat with a certain time interval. For the CAP theory, Eureka priority is satisfied by the AP condition, availability priority.

Now suppose we want to migrate the application to the cloud and deploy it in a Kubernetes cluster. At the same time, Eureka will also be deployed in the cluster, and the service discovery of the application will still use Eureka. This time, there may be a problem: the instances previously deployed based on physical machines or virtual machines are relatively stable, while the Pod in Kubernetes usually changes frequently, such as deployment resource updates and auto-scaling; these situations bring frequent updates of Pod IP, such as deleting registration information when deleting old Pods and registering again after creating new Pods. Such frequent changes with synchronization latency can lead to service consumers getting incorrect service addresses, resulting in access failures, as shown in Fig. 4.3.

A typical characteristic of a cloud-native application is that it is always in a process of change, and it needs to adapt itself through changes to meet the user's desired state. This characteristic makes it incompatible with traditional service discovery mechanisms, like two gears at different speeds that hardly work together.

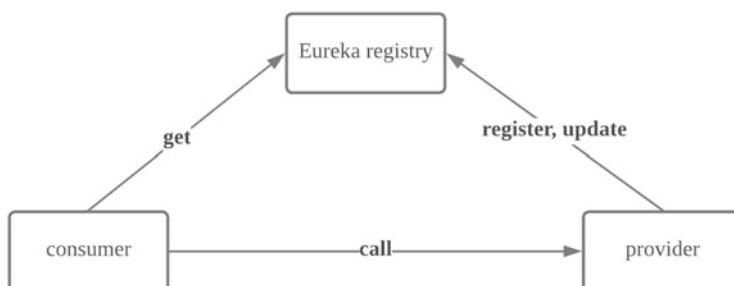


Fig. 4.2 Eureka service discovery

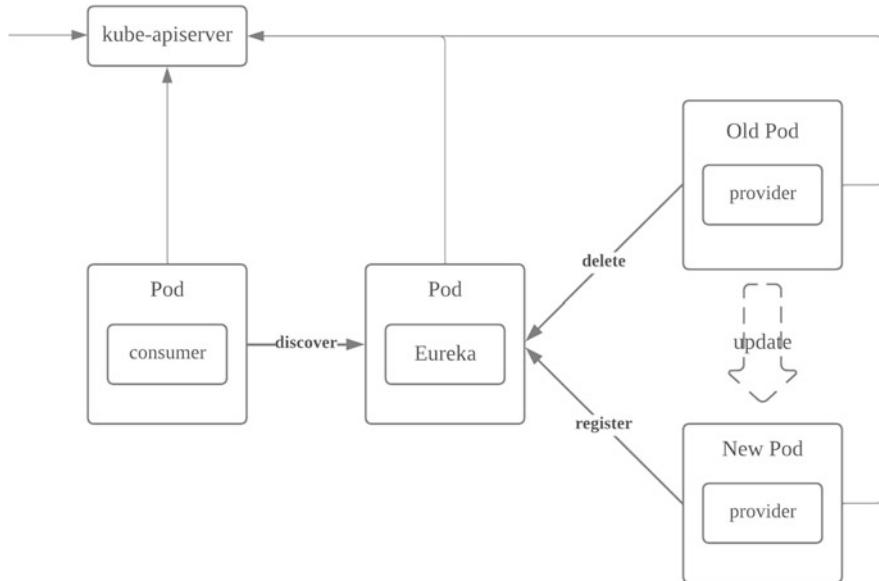


Fig. 4.3 Eureka in kubernetes

4.2.2 *Kubernetes' Service Discovery Mechanism*

Kubernetes' service discovery mechanism is mainly implemented through Service resource, DNS and kube-proxy, and in this section we'll see how it works.

Service

Kubernetes solves the problem we mentioned in Sect. 4.2.1 with Service, which is actually fixed access points assigned to Pods by Kubernetes and can be considered as an abstraction of a set of Pods. We can think of the Service object as consisting of two parts.

- Front end: name, IP address, port, and other stable and unchanging parts
- Back-end: the corresponding collection of Pods

The caller accesses the service through the front-end, which remains unchanged throughout the Service lifecycle, while the back-end may change frequently, as shown in Fig. 4.4.

With the Service as the access portal, the caller does not need to care about the Pod changes at all, but only needs to deal with the stable Service, and the problem of delayed or inconsistent service discovery is naturally solved. The controller associated with the Service continuously scans for matching Pods and updates the relationship between the Service and the Pod.

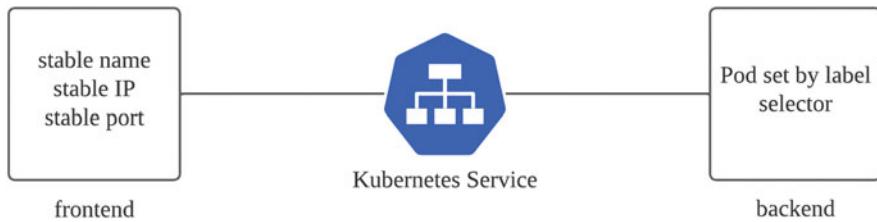


Fig. 4.4 Kubernetes service

Service Registration

Like Eureka, Kubernetes requires a service registry, which uses DNS as a service registry. Each cluster runs a DNS service, for example, the current default DNS is CoreDNS, which can be found in the kube-system namespace.

```
$ kubectl get po -n kube-system
NAME           READY   STATUS    RESTARTS   AGE
coredns-f9fd979d6-hsbsv   1/1     Running   Running   9d
9dcoredns-f9fd979d6-kzj76 1/1     Running   Running   9d
```

Each Service is registered in this DNS, and the registration process is roughly as follows.

- Submit a new Service definition to kube-apiserver.
- Create Service, assign a ClusterIP to it, and save it to etcd.
- The DNS service listens to the kube-apiserver and creates a domain record mapped from the Service name to the ClusterIP as soon as a new Service is found. This way the service does not need to actively register and relies on the DNS controller to do so.

There is a Label Selector field in the Service, and Pods that match the labels defined in the selector are included in the current Service's Pod list. In Kubernetes, the Pod list corresponds to an object named Endpoint, which is responsible for keeping a list of Pods that match the Service label selector. For example, here is a Service named reviews that has three IP addresses in its Endpoint, corresponding to three Pods each.

NAME	ENDPOINTS	AGE
reviews	10.1.0.69:9080,10.1.0.72:9080,10.1.0.73:9080	9d

The selector controller of the Service continuously scans for Pods that match the labels in the Service and then updates them to the Endpoint.

Service Discovery

To use the DNS service discovery mechanism, it is important to ensure that each Pod knows the DNS service address of the cluster. Therefore the /etc./resolv.conf files of the containers in the Pods are configured to use the cluster's DNS for resolution. For example, in the above reviews service container, the DNS configuration is exactly the address of the CoreDNS service in the cluster.

```
# reviews the DNS configuration file in the service container
user@reviews-v1-545db7b95-v44zz:~$ cat /etc/resolv.conf
nameserver 10.96.0.10
10search default.svc.cluster.local svc.cluster.local cluster.
local# Information about the CoreDNS service in the cluster
$ kubectl get svc -n
kube-system NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S)
AGEservice/kube-dns ClusterIP 10.96.0.10 <none> 53/UDP,53/TCP,9153/TCP 9d
```

What follows is a typical DNS query process. The caller initiates a query to the DNS service for the domain name (e.g., the name of the reviews service). If there is no local cache, the query is submitted to the DNS service and the corresponding ClusterIP is returned. The ClusterIP is only the IP address of the Service, not the IP address of the specific Pod providing the service, so the request also needs to access the corresponding Pod via each node in Kubernetes and has a service called kube-proxy, which creates iptables or IPVS routing rules based on the correspondence between the Service and the Pod, and the node forwards the request to the specific Pod based on these rules. As for the details, they belong to container communication, so I won't go into them here.

Service discovery for service meshes is also generally done with the help of the Kubernetes platform described above. The difference is that because of the introduction of the sidecar proxy, traffic is first hijacked by the proxy, which is implemented by modifying the iptables routing rules. The control plane of the service mesh also listens for changes in Service within the cluster to get the latest service addresses. At the load balancing level, the service mesh is usually based on Service rather than Pods, i.e., a layer of load balancing is added in front of the Service. This is also the reason why it enables service-level traffic management.

With the addition of cloud-native technologies like Kubernetes and Service mesh, service traffic management has changed a lot and become smarter and more transparent. In the next section, we will detail how to implement traffic management using Istio.

4.3 Using the Istio Service Mesh for Traffic Management

Using service meshes for traffic management may be the most appropriate option in a cloud-native. After 5 years of development, Service mesh products become maturing, and Istio is one of the most popular ones. CNCF did market research in 2020, which showed that nearly 30% of organizations are currently using Service meshes,

with Istio having a share of over 40%. The core business system developed by our team is currently deployed in a Kubernetes cluster on the cloud and uses Istio as a traffic management solution. This section will explain how to use Istio for dynamic routing and traffic management based on the team's best practice.

4.3.1 Core CRDs

Istio is a service mesh product developed by Google, IBM, and Lyft, which is officially defined as follows.

Istio is an open-source service mesh that layers transparently onto existing distributed applications. Istio's powerful features provide a uniform and more efficient way to secure, connect, and monitor services. Istio is the path to load balancing, service-to-service authentication, and monitoring—with few or no service code changes.

Istio is a typical service mesh product, and its main capabilities are in traffic management, security, and observability. Istio's traffic management function mainly relies on Envoy sidecar proxy. Traffic is first hijacked by the proxy and forwarded to application services. This is not a book dedicated to service meshes, so the relevant implementation details will not be repeated. Our main concern is how to implement traffic management through Istio. This section focuses on a few core custom resources (CRDs) related to traffic management.

Istio's traffic management capabilities are reflected in three main areas.

- Dynamic routing and traffic shifting: basic routing settings, scaled traffic slicing, etc.
- Resiliency: timeout, retry, fuse, current limit
- Traffic debugging: fault injection, traffic mirroring

The main traffic management custom resources in Istio are as follows, as shown in Table 4.1.

VirtualService

If we think of Service in Kubernetes as access points to Pods, then VirtualService is an abstract access point to Service, directing requests to the corresponding Service. A VirtualService is essentially a set of routing rules by which requests can be distributed to the corresponding service. Here are a few simple examples to illustrate its usage scenarios.

Table 4.1 Istio CRDs for traffic management

Customized resources	Explanation
VirtualService	Define routing rules to control which destination address the request is forwarded to
DestinationRule	Policy for configuring requests
ServiceEntry	Registering an external service inside the grid and managing the traffic that accesses it
Gateway	Set up load balancing and traffic shifting at the boundaries of the grid
Sidecar	Control traffic hijacking by sidecar proxies
WorkloadEntry	Registering workloads such as virtual machines to the service mesh

Unified Access Portal (Request Distribution)

The following configuration implements the function of distributing requests to different services based on URLs and works with Gateway resources for the scenario of unified request entry.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs
spec:
  hosts:
    - *.example.com
  http:
    - match:
        - uri:
            prefix: /order
      route:
        - destination:
            host: order.namespace.svc.cluster.local
```

When the target address of a user request is example.com, the request is sent to the order service if the URL has order in the prefix. Hosts field is the target address of the request, which can be a domain name, service name or IP address, etc. Istio also provides a rich set of matching conditions, which can be based on header information, methods, in addition to the paths demonstrated above, port, and other attributes.

Proportional Traffic Migration

Another common application scenario is to split and move traffic to different services (versions), such as what we usually call canary release, which can be easily implemented by VirtualService simply. You can define different types of subset (e.g., version number, platform, source), and the traffic can be flexibly distributed in

different dimensions. The distribution weight of the traffic is set by the weight keyword; let's see an example.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: vs-canary
spec:
hosts:
- order.ns.svc.cluster.local
http:
- route:
- destination:
host: order.ns.svc.cluster.local
subset: v2
weight: 10
- destination:
host: order.ns.svc.cluster.local
subset: v1
weight: 90
```

This configuration implements a simple canary release. Order service has two versions, v1 and v2, and 10% of requests are sent to v2 and the other 90% are sent to v1. The versions are defined using the subset keyword. The version is defined by the subset keyword. Subset is actually the label for a specific version, and its mapping relationship to the label is defined in another resource DestinationRule. In general, a VirtualService that references a subset must define it in the DestinationRule; otherwise, the proxy will not find the version of the service to be sent.

Timeout, Retry

VirtualService has two keywords, timeout and retries, which can be used to easily implement timeout and retry features. For example, in the following example, we set a timeout policy of 2 s for the order service. If the service does not respond after 2 s, it will return a timeout error directly to prevent the downstream service from waiting.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: vs-timeout
spec:
hosts:
- order
http:
- route:
- destination:
host: order
timeout: 2s
```

Retries can also be set via the retries keyword. The following example shows that if the order service does not return within 2 s (set by perTryTimeout), a retry will be triggered, and up to 3 times.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-retry
spec:
  hosts:
    - order
  http:
    - route:
        - destination:
            host: order
  retries:
    attempts: 3
    perTryTimeout: 2s
```

Fault Injection

Istio provides a simple fault injection feature, which is also configured through VirtualService and can be set using the fault keyword.

- Delay: Use the delay keyword to add a response delay to the service.
- Abort: Use the abort keyword to abort the service response and returns an error status code directly, such as returning a 500 error status code.

In the following example, we inject a latency fault that causes a 5 s delay in the response of the order service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-delay
spec:
  hosts:
    - order
  http:
    - fault:
        delay:
          fixedDelay: 5s
    - route:
        - destination:
            host: order
```

Traffic Mirroring

Briefly, traffic mirroring is the process of making a copy of traffic and sending it to a mirroring service in real time. This feature is especially useful when debugging online issues. In general, it is difficult to simulate the online situation in the development environment or even the staging environment. In addition to the inconsistent deployment environment, the different user usage and testing environment also make the data environment inconsistent, so many problems are difficult to be found during the development and testing process, and it is difficult to debug in the production environment, for example, the online log level is set high to avoid generating a lot of debug logs and wasting resources. This is where traffic mirroring comes into play. We can deploy a service identical to the production environment and set the log level to debug only. When a user accesses, a copy of the request is sent to the mirroring service, and we can view the debug information in the mirroring service's logs to find the problem.

Implementing traffic mirroring in Istio is very simple by a mirror field of VirtualService. The following example adds mirroring to the order service with a mirrorPercent of 100%; i.e., all requests sent to version v1 are copied and sent to mirror service v2. Of course, a subset of both versions still needs to be defined in the DestinationRule.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-mirror
spec:
  hosts:
    - order
  http:
    - route:
        - destination:
            host: order
            subset: v1
            weight: 100
        mirror:
            host: order
            subset: v2
  mirrorPercent: 100
```

VirtualService is the most important resource in Istio, providing flexible and powerful traffic management configuration capabilities. In essence, VirtualService is a series of routing rules that tell requests which destination address they should go to.

DestinationRule

DestinationRule is another important resource that generally works with VirtualService. In short, VirtualService determines which destination the request goes to, and DestinationRule determines how the request should be handled once it reaches a destination.

- **Subset:** Divide the requests by defining subsets to be used with VirtualService in different dimensions.
- **Traffic policy:** Define how the request will be handled once it reaches the target address, such as load balancing, connection pool, etc.
- **Circuit breaking:** Set the circuit breaker by error detection.

Subset

The following example shows the content of defining a subset, which is used in conjunction with the VirtualService by version routing example above. The subset added to the destination field of the VirtualService is defined here, and the corresponding version label is set in the Pod, so that requests can be distributed to the corresponding target Pod.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-order
spec:
  host: order
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Definition of Traffic Policy

DestinationRule is also responsible for defining traffic policies, such as load balancing policy, connection pooling, etc. The following example shows the process of defining a traffic policy with a random pattern for load balancing, a TCP connection pool size of 100, and a timeout of 2 s.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-policy
spec:
```

```

host: order.ns.svc.cluster.local
trafficPolicy:
  loadBalancer:
    simple: RANDOM
  connectionPool:
    tcp:
      maxConnections: 100
      connectTimeout: 2s

```

Circuit Breaker

The term circuit breaker is used in many places, for example, a stock circuit breaking is when the exchange suspends trading in a particular stock. At the service governance level, a circuit breaking is when a service fails waiting for a period of time before trying to connect. If the service is recovery, then the circuit breaker is closed and the request resumes accessing the upstream service; otherwise, the circuit breaker continues to be kept open.

Circuit breaking is also a form of service degradation that is a bit smarter than timeout and retry, automatically detecting the availability of a service and allowing the service to return from a broken flow to a normal state. Unlike timeouts and retries, it is implemented by setting the outlierDetection field in the DestinationRule, rather than in the VirtualService. The following example shows a basic configuration process.

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-cb
spec:
  host: order.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      consecutive5xxErrors: 5
      interval: 5m
      baseEjectionTime: 15m

```

The consecutive5xxErrors field indicates that 5 consecutive 5xx errors will trigger a circuit breaking. baseEjectionTime is used to configure the time when the instance with errors is expelled from the load balancing pool, which is 15 mins in this case. Interval is actually the timeout clock, which indicates that every 5 mins the breaker will become half-open and retry to access the upstream service. The circuit breaking mechanism in Istio is not as complete as that in Hystrix, so there is no way to define the logic for handling the error after the circuit breaking. Also, be careful to do more tests on error determination so as not to “miss” requests with status code 5xx that want to return normally.

ServiceEntry

ServiceEntry provides a way to manage access to external service traffic. It is equivalent to an abstract proxy for external services within the service mesh, through which it is possible to register external services in the service registry, and other services within the service mesh can access it as if it were an internal service.

By default, services within Istio are able to access external services directly, so why do you need to define ServiceEntry? The reason is simple. Although internal requests can go out directly, they cannot use the capabilities provided by Istio. So you have no way to manage the traffic of these accesses to external services. With ServiceEntry, we can bring the traffic that accesses external services under Istio's control as well.

The following example defines a ServiceEntry for the external service [ext.example.com](#). To control this traffic, you only need to define the corresponding VirtualService and DestinationRule.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: se
spec:
  hosts:
  - ext.example.com
  ports:
  - number: 80
    name: http
    protocol: HTTP
  location: MESH_EXTERNAL
  resolution: DNS
```

Gateway

There are two types of gateways provided in Istio, the Ingress gateway and the Egress gateway, which correspond to the above two application scenarios. You can think of a Gateway simply as a load balancer that runs at the edge of the mesh when receiving inbound or outbound traffic. Note that the Gateway in Istio is only used to define basic information about a gateway, such as the public port, the type of protocol to be used, etc. The specific routing rules are still defined in the VirtualService and matched by the gateways field. This design is different from Ingress in Kubernetes, which separates the definition of gateways and routing information to achieve loose coupling.

The most common scenario for a Gateway is to act as a unified access portal for outside traffic and then distribute the requests to different services of the application. The following example defines an entry gateway that listens on port 80 for access to [example.com](#) and then directs requests to different services on the back-end based on URL prefixes.

```

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: gw-ingress
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - *.example.com
  --
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs
spec:
  gateways:
  - gw-ingress
  hosts:
  - example.com
  http:
  - match:
    - uri:
        prefix: /order
    route:
    - destination:
        host: order.namespace.svc.cluster.local

```

The selector field in Gateway defines a selector for the gateway. In this example, we are using the ingressgateway that comes with Istio, which is essentially a Service of type LoadBalancer in Kubernetes that accomplishes the corresponding functionality through Envoy. You can also use your own implementation of the gateway. Note that if a VirtualService is multiplexed by both a gateway and a service, then both the gateway name and the “mesh” should be filled in the gateways field.

Sidecar

Sidecar is a custom resource that controls the behavior of the sidecar proxy within the service mesh, which is why it is called Sidecar. By default, the Envoy proxy in Istio listens to traffic on all ports and forwards traffic to any service within the service mesh. Configuring Sidecar resources can change this default behavior as follows.

- Modify the set of ports and protocols that the sidecar proxy listens on.
- Restrict the services that can be accessed by the sidecar proxy.

Sidecar can help fine-tune the behavior of sidecar proxys or use different policies for different proxys. For example, the following example is configured so that the sidecar proxy under the default namespace only listens to inbound traffic on port 9080.

```
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: sidecar-test
  namespace: default
spec:
  ingress:
  - port:
    number: 9080
    protocol: HTTP
    name: http
```

WorkloadEntry

Istio involved the WorkloadEntry in version 1.6, which brings non-Kubernetes workloads into the management of the service mesh. WorkloadEntry needs to be used with ServiceEntry. It defines the IP address of a workload and an app label; ServiceEntry fills in the same label in its own selector, enabling co-working with WorkloadEntry.

The following code shows how to create a WorkloadEntry for a virtual machine with IP address 2.2.2.2 and specify the label of the application as vmapp.

```
apiVersion: networking.istio.io/v1alpha3
kind: WorkloadEntry
metadata:
  name: vmapp
spec:
  serviceAccount: vmapp
  address: 2.2.2.2
  labels:
    app: vmapp
  instance-id: vm1
```

The corresponding ServiceEntry configuration code is as follows, which defines in the workloadSelector field that the application label to be associated is vmapp.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: vmapp
spec:
  hosts:
  - vmapp.example.com
  location: MESH_INTERNAL
  ports:
```

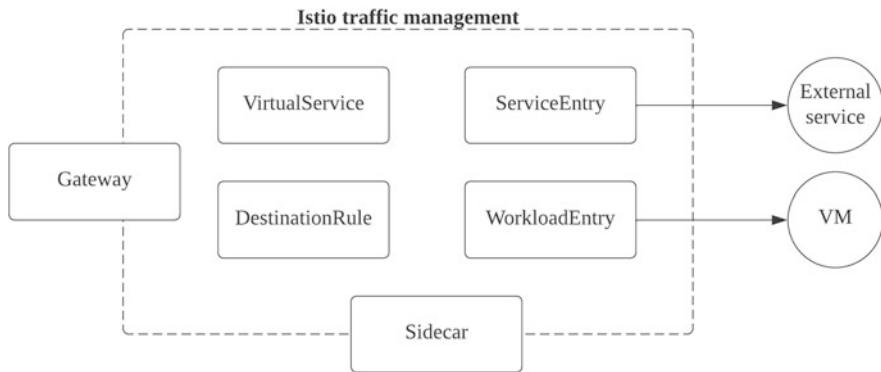


Fig. 4.5 Istio traffic management

```
- number: 80
  name: http
  protocol: HTTP
  targetPort: 8080
resolution: DNS
workloadSelector:
  labels:
    app: vmapp
```

These are custom resources related to traffic management in Istio. As you can see, with these several resources, we can easily manage the traffic within the service mesh, outside the service mesh, and at the service mesh boundary, as shown in Fig. 4.5.

4.3.2 *Istio-Based Traffic Management Practices*

Our core business system we developed is an online video ad delivery platform. Starting from 2017, the team migrated the original Ruby-based monolithic application into a Golang-based microservice application. In the following years, the system gradually completed technical adoption such as containerization, onboarding AWS, and service mesh adoption, and finally became a cloud-native application. The application includes nearly 100 microservices, over 2000 gRPC APIs and handles millions of requests per day. As the number of services grows, the need for governance and traffic management becomes stronger and the service mesh becomes the preferred solution.

Introduction of Service Mesh as Traffic Management Infrastructure

There are several main reasons for using a service mesh for traffic management.

- Lack of traffic management and service governance capabilities: When performing microservice adoption, we will rely on the typical “stranded” approach, i.e., quickly build a service, then move the traffic to the new system to gradually replace the old system in an incremental manner, and finally complete the migration of the entire traffic. The biggest advantage of this approach is that it can quickly bring microservices online and provide business features, which is suitable for teams that want to quickly upgrade their technology stack. However, the drawback of this solution is also obvious: it is a bottom-up construction way, lacking a unified and common microservice architecture or common middleware, where each service gives priority to implementing business logic and then gradually extracts common capabilities to build the architecture. Therefore, the application lacks service governance capabilities, and each service will choose different design solutions to achieve nonfunctional requirements according to its own needs. This causes code duplication and human resource waste to some extent. With more and more services, this inconsistency also brings many difficulties to maintenance. In this case, service mesh became the first option to solve this problem.
- Lack of tracing and service monitoring capability: The most important feature of our system is that the business is very complex, communication between services is frequent, and many business transactions require the participation of multiple services to complete. This makes it very difficult to debug when problems occur. I believe many developers have encountered similar scenarios, for example, a user has a problem when using the system, and the developer of service A finds after debugging that there is an error when calling service B, while the developer of service B finds after investigation that there is a problem when calling service C. The longer the calling trace, the more difficult it is to find the root cause of such a problem. Without a tracing system, through the logs to troubleshoot the problem, the efficiency will be very low. In addition, we do not have multi-faceted monitoring means for the service, and the existing monitoring indicators and other contents are still only based on the original single application. We urgently need to build a whole set of monitoring means to observe the status of the service in all aspects. The integration capability provided by the Service mesh in terms of observability precisely gives us convenience.
- Single technology stack with no historical burden: Our microservice application is not a heterogeneous system (technologies such as serverless computing are also being gradually introduced) and unlike many companies in the industry that already use microservice frameworks like Spring Cloud, there is no need to consider the cost of migrating from a framework to a service mesh. Also, we have no additional burden to consider for service registration and service discovery integration. The services in the application are all containerized and deployed

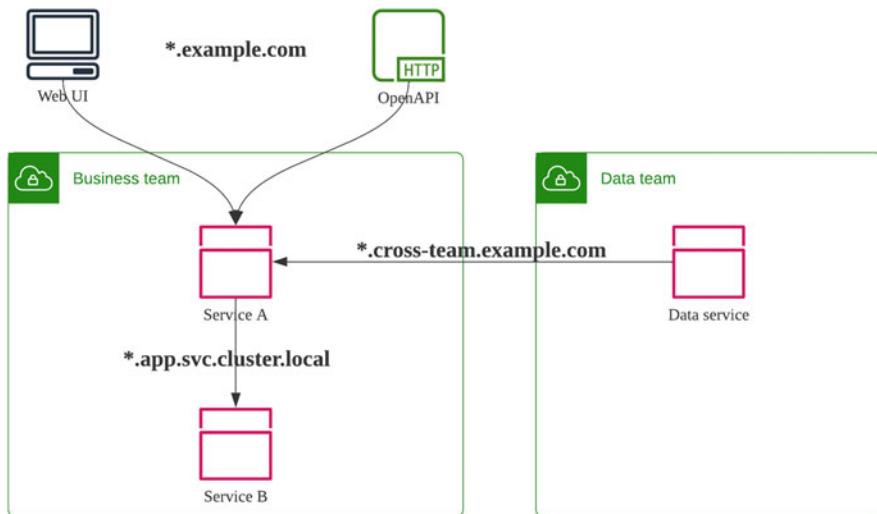


Fig. 4.6 Traffic types

in a Kubernetes cluster, which makes the implementation of a service mesh a snap.

For these reasons, we have chosen to use a service mesh to improve the system architecture and meet the existing needs for service governance and traffic management. Our advice on whether you need to use a service mesh is to start with the requirements and evaluate whether it can solve your pain points, while also considering the cost of access. If you already have a relatively mature service governance solution in your system or don't have many pain points, there is no need for a technology stack update; new technologies not only have costs but are also likely to bring new problems.

Traffic Scenarios

The application generates the following three types of traffic.

- **External traffic:** It is actually the requests from outside the company, from the public network through the gateway into the application, such as example.com in Fig. 4.6. The application exposes two kinds of external traffic to the outside, one is the web-UI requests, customers use our application through the web page, and then send requests to the back-end microservices. The other is the external open API, also known as OpenAPI, which is a series of RESTful APIs. Customers can call the APIs directly in their own system to use the features provided by the system.

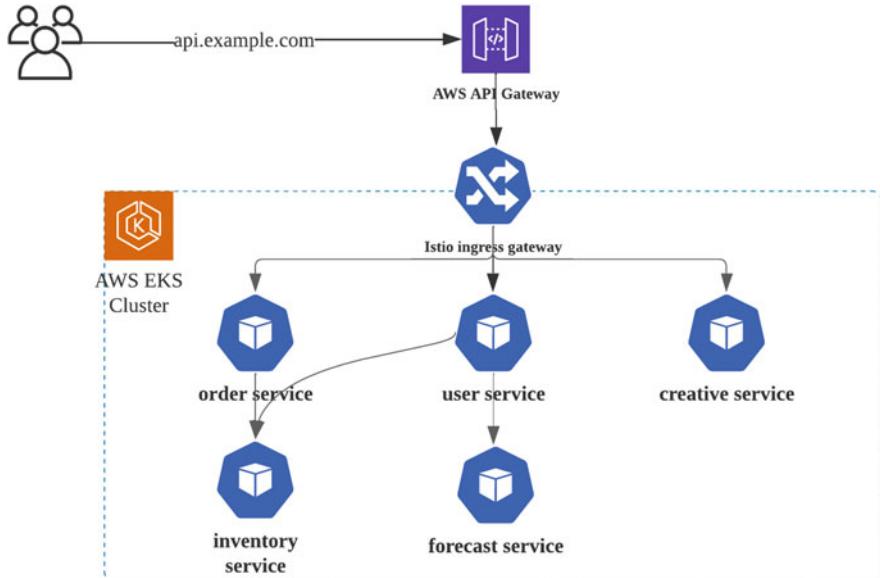


Fig. 4.7 Istio ingress gateway

- Internal traffic: Traffic between services within a microservice application, such as `app.ns.svc.cluster.local` in Fig. 4.6. In general, this is the largest source of traffic for microservice applications, as there is much more interaction between services than between services and the outside world. Managing the communication between services is also the most important responsibility of the service mesh. Obviously, the complexity of internal traffic increases as the number of services increases because the network topology of service interactions becomes more complex. It is also usually the lack of management of this part of the traffic in the system that makes it necessary to consider introducing a service mesh.
- Cross-system traffic: This traffic comes from calls to the microservice application from other systems within the company, such as `cross-team.example.com` in Fig. 4.6. The data team has a data service that needs to call the business team's microservice application, and this request is external to the microservice application and generates internal traffic for the entire company or organization.

The above three types of traffic can be essentially categorized as east-west traffic and north-south traffic. In general, there are similar request scenarios in each application that generate the corresponding traffic.

Istio's Deployment and Network Topology

Once we understand the source of the request, let's take a look at the network topology of the application. Figure 4.7 shows a relatively complete request trace and

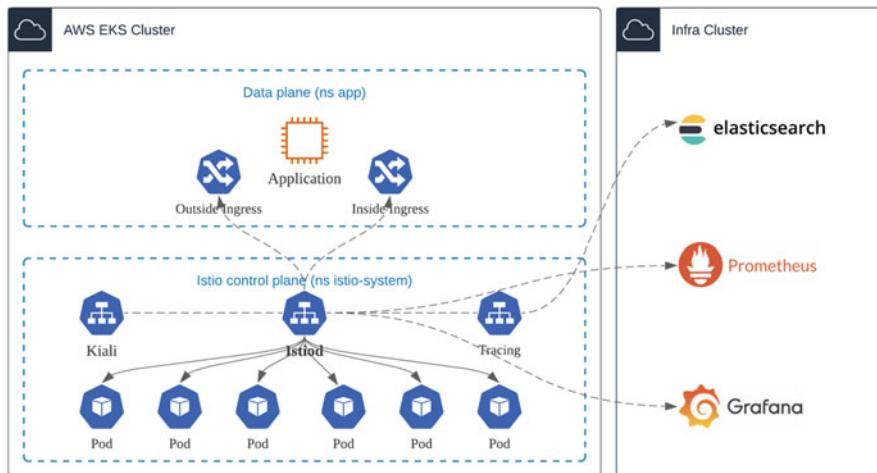


Fig. 4.8 Istio deployment

the main network devices. Taking external traffic as an example, the user invokes the system's OpenAPI and the request first passes through the AWS API gateway where authentication-related operations are performed and then is forwarded to the Kubernetes cluster where the application resides (AWS EKS cluster). There is an Istio entry gateway at the application's boundary, which is a business gateway responsible for forwarding requests to the corresponding application service based on the URL prefix.

Take another look at the deployment structure of Istio, as shown in Fig. 4.8.

By default, Istio is installed under the Istio-system namespace of the Kubernetes cluster, and the core is the Istiod, which we usually refer to as the control plane. To avoid single-point problems, we deployed six instances of Istiod for load balancing.

Another important component is the Istio ingress gateway, which is essentially a Service of LoadBalancer in Kubernetes. We have deployed two different ingress gateways in order to be able to manage external traffic and cross-system traffic within the company separately. If you need to use observability-related features, you can also deploy the corresponding components, such as Istio's monitoring tool Kiali. Other metrics collection, log collection, and other functions can be directly implemented using existing services and configured to interface.

Sidecar does not need to be manually added to the Deployment configuration, but simply adds the `istio-injection = enabled` label to the namespace where the application is located to automatically inject the Sidecar proxy when the Pod startup. At this point, a service mesh is basically built, and the application has the features provided by the service mesh; all that remains is to use these features in a declarative configuration and implement traffic management as required.

Routing Configuration Example

Once the service mesh is built, the next step is to use it for traffic management. The first problem to solve is how to route through the service mesh, i.e., forward requests to the correct address with the service mesh, which is the most basic usage scenario. We will use the two scenarios described above, external traffic and internal traffic, to show how to do the routing configuration.

External Traffic Configuration

For external traffic, it is generally necessary to create two Istio objects: an ingress gateway and a virtual service. The ingress gateway is used to define the ingress of the requested traffic, as well as the protocol, port, and other information to listen on, while its corresponding virtual service is used to implement specific routing rules.

Still using the above-mentioned content as an example, the user sends a request with the URL api.example.com/orders/1 to get order number 1. Obviously, this API is provided by the order service. In order for the request to find the order service, we first define an entry gateway object outside-gateway, which is responsible for listening to HTTP requests for the *.example.com domain, as defined below.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: outside-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*.example.com"
```

After defining the gateway, you can create it with the kubectl apply command, and then you can see the generated object in the cluster.

```
$ k get gateway
NAME          AGE
outside-gateway  6s
```

As we introduced earlier, the ingress gateway only provides the definition of the access portal; the specific routing rules need to be implemented through a virtual service. Next, we define a virtual service corresponding to the gateway.

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-for-outside-gateway
spec:
  hosts:
    - "*.example.com"
  gateways:
    - outside-gateway
  http:
    - match:
        - uri:
            prefix: /orders
      route:
        - destination:
            host: orders.app.svc.cluster.local
            port:
              number: 9080

```

The virtual service is bound to the gateway via the gateways field, which is set to the corresponding gateway name. The hosts field indicates that the requests whose domain name is `*.example.com` are routed by it. Next we use the match field to make a simple matching rule, i.e., requests with the word `orders` in the URL, their destination address is the host address in the destination field, which is the name of the specific order service. That is, requests that match this rule are forwarded to port 9080 of the order service. At this point, a request from the outside can be routed through the ingress gateway to a microservice inside the cluster. The above code is only for the service configuration part. In principle, microservice applications need to define routing rules in this virtual service as long as they have an externally exposed API, but of course, multiple configurations can be divided to handle each service separately.

Service-to-Service Communication (Internal Traffic) Configuration

The configuration of service-to-service communication is relatively simple, requiring no ingress gateway involvement and only the creation of virtual service objects. If there are no traffic management requirements or even the need to create virtual services, requests are forwarded based on Kubernetes' service discovery mechanism. If there is a traffic policy requirement, a DestinationRule can be added. The following code is an example of the most basic routing configuration, where the virtual service `vs-order` does not do any traffic management and simply points the request to port 9080 of the order service.

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-order
spec:

```

```
hosts:
- orders
http:
  route:
    - destination:
        host: orders
        port:
          number: 9080
```

It would be a bit of an overkill to use Istio only for such basic routing. In real application scenarios, we will definitely use more advanced features. For example, in the following example, we have implemented a feature that distributes requests by type by defining a virtual service and a destinationrule object. Requests with version = v2 in the header will be forwarded to the v2 version of the service. The version of the service is defined in the destinationrule as a subset of information, and both versions of the service are deployed in the cluster with the corresponding label information.

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
...
spec:
  hosts:
  - orders
  http:
    - match:
        - headers:
            version:
              exact: v2
      route:
        - destination:
            host: orders
            subset: v2
    - route:
        - destination:
            host: orders
            subset: v1
  ---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: orders
spec:
  host: orders
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
```

The above is a typical example of request distribution, which can be extended with a few modifications, for example, by adding a weight field to implement the proportional distribution of traffic, such as canary release.

As you can see, Istio's traffic management is implemented completely in a cloud-native way, with declarative configurations to define the control logic, without any modifications at the application level at all. These configurations can be merged into a unified management in the Helm Chart of the deployed service as needed. It is important to note that Istio officially recommends that different services should manage their own configurations so that route definitions do not conflict. However, in reality, service meshes are generally maintained by dedicated ops teams, and it makes sense for them to manage them, as not every business development team is familiar with service mesh technology. Configuration errors can lead to request routing exceptions, so our experience is that configuration maintenance is determined by the team's own.

The above describes the routing configuration, and we will cover service governance-related content such as timeouts and retry in Sect. 4.4. In addition, we have accumulated some experience in observability, with applications in logging, metrics, and tracing, which will be covered in detail in Chap. 7 of this book.

4.3.3 *Common Adoption Problems and Debugging*

Troubleshooting is a common job faced no matter which technology is used. For Service mesh, sidecar proxy makes the number of requests forwarded increase, and debugging difficulty increases accordingly. This section will introduce how to debug the service mesh and solve the problems that arise during use based on our experience in the field.

Summary of Common Problems

The author summarizes several problems easily encountered during the adoption of the service mesh, as follows.

There Are Requirements and Restrictions on the Application

Although a service mesh like Istio is transparent to the application, it does not mean that there are no restrictions when using it. If the application is deployed on a Kubernetes cluster, it is important to note that Istio has some requirements for Pod setup. The author lists a few of them that are prone to problems.

- Pods must belong to some Kubernetes Service. This is easy to understand because request forwarding is based on Service, not on Pods. Also, if a Pod belongs to multiple Services, services of different protocols cannot use the same port

number. It is also easy to understand that there is no way for the Sidecar proxy to send requests to the same port to different services.

- Pods must add app and version labels. Istio officially recommends that these two labels should be configured explicitly, the app label (application label) to add contextual information in distributed tracing and the version label (version label) to distinguish between different versions of the service. It is also easy to understand that features like blue-green deployments and canary release need to split traffic based on version.
- Port naming rules. Before Istio 1.5, the port names of Service had to be prefixed with the protocol name. This is because Istio must know the protocol if it wants to manage Layer 7 traffic, and then implement different traffic management functions based on the protocol. The Kubernetes resource definition does not contain Layer 7 protocol information, so it needs to be explicitly declared by the user. However, this problem has been solved in later versions, and Istio can already automatically detect both HTTP and HTTP/2 protocols. Also, in Kubernetes 1.18 and above, it is possible to add the appProtocol: <protocol> field to the Service to enable protocol declaration.
- Non-TCP protocols cannot be proxied; Istio supports any TCP-based protocol, such as HTTP, HTTPS, gRPC, and pure TCP, but non-TCP protocols, such as UDP requests, cannot be proxied. Of course, the requests will still work; they just won't be intercepted by Sidecar.

In addition to these requirements for Pods, there are some other restrictions, such as not being able to occupy Istio's default ports, most of which start with 150XX and require attention.

503 Errors

It is a very common problem to get 503 errors after using the service mesh, and it is difficult to find the root cause. This is because the original direct connection request of service A to access service B will be forwarded twice more due to using Sidecar proxy, and it is difficult to find the breakpoint where the request has problems. An important field in Envoy logs is RESPONSE_FLAGS, which will show the corresponding error identifier when the request is unexcepted. For 503 errors, there are mainly the following kinds of identification.

- UH: There are no healthy hosts in the upstream cluster.
- UF: Upstream service connection failed.
- UO: Upstream service circuit breaking.
- LH: Local service health check failed.
- LR: Local connection reset.
- UR: Upstream remote connection reset.
- UC: Upstream connection closed.

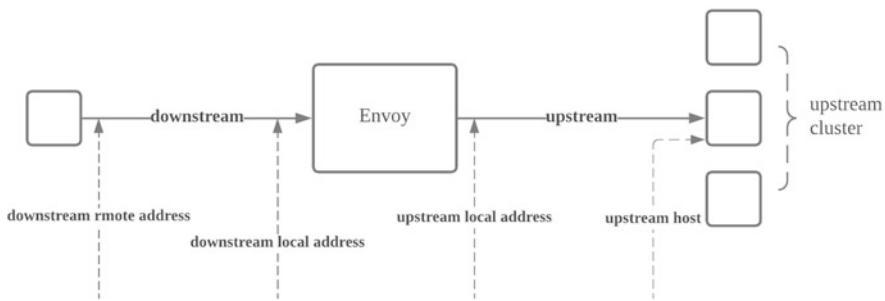


Fig. 4.9 Envoy five-tuple context

For example, in the example below, the error flag in the log is UH, which means there are no healthy hosts in the upstream cluster. After investigation, it was found that the startup failed because of a problem with the order service deployment, i.e., there was no Pod available in the service endpoint list.

```
[2021-05-11T08:39:19.265Z] "GET /orders/1 HTTP/1.1" 503 - "UH" 0 178 5 5 "-" ...
```

In general, we can understand the general cause of the error by the error flag. If we cannot find the root cause of the problem, we can further analyze it by using the upstream and downstream information in the log. Figure 4.9 shows the five-tuple model.

These five fields (downstream remote address, downstream local address, upstream local address, upstream host, and upstream cluster) are very important information in Envoy logs and can be used as the basis for us to determine the request breakpoint.

The Route Is Not Configured to Cause 404 Errors

If the request has a 404 error and NR appears in the error flag of the log, it means that there is a problem with the routing configuration. The first thing to check is whether there is a problem with the configuration of the virtual service and the corresponding destination rule. There is also a possibility that we encountered a problem with the order in which the configuration is sent down. For example, we have defined in the virtual service that the request is to be routed to version v1, and a subset of that version is defined in the corresponding destination rule. If both objects are created at the same time via kubectl apply, there may be a problem. The reason for this is that Istio only guarantees final consistency when issuing configurations, and in the process it is likely that the configuration of the virtual service was issued first, while the destination rule it depends on is not yet in effect, which can lead to requests reporting errors over time, as shown in Fig. 4.10.

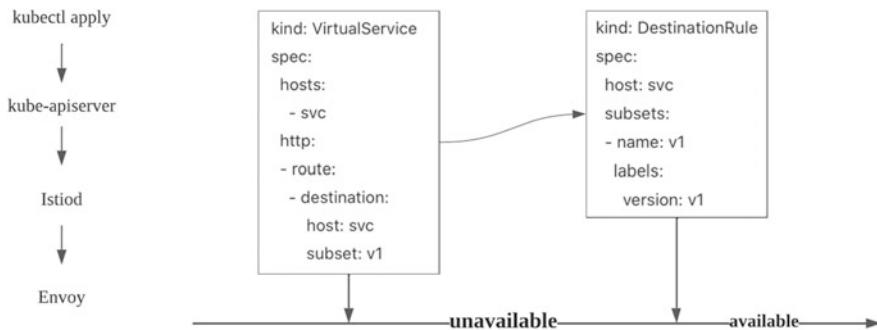


Fig. 4.10 Istio configuration refresh

This problem is self-healing, however, and will return to normal once the configuration is all in effect. It is also very simple to avoid this kind of problem by letting the dependencies finish creating first, as follows.

- When creating: Create the destination rule first, and then create the virtual service that references the subset.
- When deleting: delete the virtual services that have references to the subset first, and then delete the destination rules.

Timeout Nesting Problem

Istio provides resiliency mechanisms like timeouts, retries, and is transparent to the application. That is, the application does not know if the Sidecar proxy is using such features. Normally, this is not a problem, but if you add a feature like timeout to your application as well, you need to be aware that improper settings may cause conflicts, i.e., timeout nesting problems. For example, you set a timeout within the application that has a timeout of 2 s for API calls to other services. After setting up the service mesh, you add another timeout of 3 s to the virtual service and set a retry. Once the request response exceeds 2 s, the timeout mechanism in the application starts working, which causes the configuration in the virtual service to lose its effect. Although Istio provides several failure recovery features, in principle the application still needs to handle failures or errors and take appropriate degradation actions. For example, when all instances in a load balancing pool fail, the Sidecar proxy returns an “HTTP 503” error. The application needs to implement fallback that can handle this error, such as returning a friendly error message, or going to another page, etc.

The problems that occur in the actual development are much more than those listed, in addition to the usual attention to sum up the experience, through the logs and monitoring analysis, but also with the help of debugging tools to complete the troubleshooting, the following specific description.

Debugging Tools and Methods

Istio provides feature-rich command line tools to debug the control plane and data plane.

Istioctl Command Line Tool

Istio has been making improvements to the command line tool `istioctl` in the last few releases to provide more useful features. It can be found in the `bin` directory after downloading the Istio installer and can be used without installation. The current (based on version 1.9.2) command line has the following features, and we pick a few of the more important ones to introduce.

Istio configuration command line utility for service operators to debug and diagnose their Istio mesh.

Usage:

```
istioctl [command]
```

Available Commands:

```
analyze      Analyze Istio configuration and print validation messages
authz       (authz is experimental. Use `istioctl experimental authz`)
bug-report   Cluster information and log capture support tool.
dashboard    Access to Istio web UIs
experimental  Experimental commands that may be modified or deprecated
help         Help about any command
install      Applies an Istio manifest, installing or reconfiguring Istio on a cluster.
kube-inject   Inject Envoy sidecar into Kubernetes pod resources
manifest     Commands related to Istio manifests
operator     Commands related to Istio operator controller.
profile      Commands related to Istio configuration profiles
proxy-config Retrieve information about proxy configuration from Envoy [kube only]
proxy-status  Retrieves the synchronization status of each Envoy in the mesh [kube only]
upgrade     Upgrade Istio control plane in-place
validate     Validate Istio policy and rules files
verify-install Verifies Istio Installation Status
version      Prints out build version information
```

- (a) `istioctl proxy-status`: This command is used to check the synchronization status of the Sidecar proxy configuration. The configuration written by the user needs to be sent down to the proxy via Istiod, which requires a process, especially when the cluster is relatively large. This command is used to check whether the configuration has been synchronized to the Envoy proxy. The following output results will indicate that the configuration has been synchronized to all proxys. If the NO SENT or STALE keyword appears, it means that there is a problem with the distribution.

```
$ bin/istioctl proxy-status
NAME          CDS      LDS      EDS      RDS
details-v1-xxx.default  SYNCED  SYNCED  SYNCED  SYNCED
productpage-v1-xxx.default  SYNCED  SYNCED  SYNCED  SYNCED
reviews-v1-xxx.default    SYNCED  SYNCED  SYNCED  SYNCED
...
...
```

Alternatively, you can directly add the specific Pod name after the command to verify that its Sidecar proxy configuration is consistent with Istiod.

```
$ bin/istioctl ps productpage-v1-6b746f74dc-6wfbs .
defaultClusters Match
--- Istiod
Listeners+++ Envoy Listeners
...
Routes Match (RDS last loaded at Tue, 11 May 2021 16:39:29 CST)
```

(b) `istioctl proxy-config`: The problem of out-of-sync configuration is actually rare, and most of the problems are still due to configuration errors. Through `proxy-config`, you can view the details of the configuration, and through the subcommands, you can view the cluster, endpoint, route, listener, and other information, respectively. The following output shows the Pod's routing configuration. Using JSON format to output the information can show the information more comprehensively.

```
$ bin/istioctl proxy-config route productpage-v1-6b746f74dc-6wfbs .
defaultNAME DOMAINS MATCH VIRTUAL
SERVICE80 httpbin.org /*80
istio-
ingressgateway.istio-system /*
80 tracing.istio-system
/*8000 httpbin
/*9080 details
/*9080 productpage /*
...
...
```

(c) `istioctl verify-install`: This command verifies the installation status of Istio and ensures that Istio is running properly. It is recommended to execute this command after the installation is completed.

```
$ bin/istioctl
verify-install1 Istio control planes detected, checking --revision "default" only
...
✓CustomResourceDefinition: destinationrules.networking.istio.io.istio-system checked successfully
✓CustomResourceDefinition: envoyfilters.networking.istio.io.istio-system checked successfully
✓CustomResourceDefinition: gateways.networking.istio.io.istio-system checked successfully
✓CustomResourceDefinition: serviceentries.networking.istio.io.istio-system checked successfully
✓CustomResourceDefinition: sidecars.networking.istio.io.istio-system checked successfully
✓CustomResourceDefinition: virtualservices.networking.istio.io.istio-system checked successfully
...
```

- (d) `bin/istioctl analyze`: This command is also useful to help analyze what configuration issues are present in the service mesh. For example, the analyze result below indicates that the Sidecar proxy version in the current Pod is not the same as the one in the injected configuration, and suggests that it may be due to the upgraded control plane.

```
$ bin/istioctl analyze
Warning [IST0105] (Pod details-v1-79f774bdb9-wvrj2.default) The image of the Istio proxy running on the pod does not match the image defined in the injection configuration (pod image: docker.io/istio/proxyv2:1.9.2; injection configuration image: docker.io/istio/proxyv2:1.8.1). This often happens after upgrading the Istio control-plane and can be fixed by redeploying the pod.
```

Control Plane Self-Checking Tool—ControlZ

Istio provides a series of dashboard to query the operation of the service mesh in a visual way. We can open the corresponding dashboard via the `dashboard` parameter in the command line tool.

```
$ bin/istioctl
dAccess to Istio web UIs
```

```
Usage:
istioctl dashboard [flags]
istioctl dashboard [command]
```

```
Aliases:
dashboard, dash, d
```

```
Available Commands:
controlz Open ControlZ web UI
envoy Open Envoy admin web UI
grafana Open Grafana web UI
jaeger Open Jaeger web UI
kiali Open Kiali web UI
prometheus Open Prometheus web UI
zipkin Open Zipkin web UI
```

The screenshot shows the Istio ControlZ interface. At the top, there's a header bar with the title "Istio ControlZ [/usr/local/bin/pilot-discovery - 10.1.0.74]" and a logo of a sailboat inside a circle. Below the header is a sidebar on the left containing links: ControlZ, Logging Scopes, Memory Usage, Environment Variables, Process Info, Command-Line Arguments, Version Info, Metrics, and Signals. The main content area has a heading "Istio ControlZ" and a message "Make a selection in the left sidebar to inspect & control aspects of this process." To the right of this message is a table showing real-time process statistics:

Process Name	/usr/local/bin/pilot-discovery
Heap Size	54,503,680 bytes
Num Garbage Collections	2,515
Current Time	5/12/2021, 4:33:14 PM
Hostname	istiod-5477bf7556-vplvp
IP Address	10.1.0.74

At the bottom of the main content area is a blue button labeled "Terminate Process".

Fig. 4.11 Istio controlZ page

As you can see from the output, most of the dashboards are related to observability tools, such as Prometheus, Grafana, etc. We skipped these general tools, and here we only introduce the ControlZ self-test page related to debugging.

The ControlZ is essentially a summary page of information about the Istio control plane, which can be opened with the command `bin/istioctl d controlz <istiod pod name>.istio-system`, as shown in Fig. 4.11. The home page shows information such as Process Name, Heap Size, Num Garbage Collections, Current Time, Hostname, and IP Address, and is refreshed in real time. The information is updated in real time and provides a basic understanding of Istio's operational status. One of the more useful options is to modify the output level of the component logs in the Logging Scopes option.

Debugging Envoy Proxy

As mentioned above, ControlZ is a self-checking page for the control plane, and Istio also provides the Envoy admin page as a tool for checking information on the data plane. The page shows the currently available management APIs for Envoy, with hyperlinks to GET requests; you can directly view the returned results, and the other interfaces need to be executed by sending POST requests. The command `bin/istioctl d envoy <pod name>. <namespace>` can open Envoy admin page, as shown in Fig. 4.12.

Command	Description
certs	print certs on machine
clusters	upstream cluster status
config_dump	dump current Envoy configs (experimental)
contention	dump current Envoy mutex contention stats (if enabled)
cpuprofiler	enable/disable the CPU profiler
drain_listeners	drain listeners
healthcheck/fail	cause the server to fail health checks
healthcheck/ok	cause the server to pass health checks
heapprofiler	enable/disable the heap profiler
help	print out list of admin commands
hot_restart_version	print the hot restart compatibility version
init_dump	dump current Envoy init manager information (experimental)
listeners	print listener info
logging	query/change logging levels
memory	print current allocation/heap usage

Fig. 4.12 Envoy admin page

First, the most commonly used option is still logging. The logging level of production environment is usually set higher, so if you want to view Sidecar's request information, you need to modify the logging level. For example, you can set the output logging of Envoy by the following command and then enter the corresponding container of Envoy to view the detailed debugging log related to the request.

```
$ curl -X POST http://$URL:15000/logging?level=debugactive loggers: admin: debug aws: debug assert: debug backtrace: debug ...
$ k logs -f productpage-v1-6b746f74dc-6wfb -c istio-proxy[2021-05-11T08:38:41.625Z] "GET /productpage HTTP/1.1" 200 - "-" 0 4183 28 28 "192.168.65.3" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.128 Safari/537.36" "a149684d-48ba-9b1c-8e32-91a329fc60b" "localhost" "127.0.0.1:9080" inbound[9080] | 127.0.0.1:34380 10.1.0.70:9080 192.168.65.3:0 outbound_9080_productpage.default.svc.cluster.local default...
```

The proxy-config command allows you to view Envoy's configuration information, and you can also find the corresponding interface in the page. For example, we can directly click config_dump to view the complete configuration, but also through the clusters and listeners option to view the corresponding configuration.

In addition, we can also find two performance-related interfaces: cpuprofiler and heapprofiler, which can be used to turn on the CPU and memory monitoring switches when performance problems occur, for example, by executing the following command, and then the corresponding prof file will be generated in the default directory. Once you have this file, you can check the specific CPU usage with the help of performance analysis tools such as pprof.

```
$ curl -X POST http://localhost:15000/cpuprofiler\?enable\=yOK
```

This section summarizes some relatively common failures and debugging methods based on the author's practice. The official Istio documentation has special O&M-related materials that list more comprehensive fault checking-related practices.

4.4 Improving Application Fault Tolerance with Istio

Fault tolerance is often called resilience and refers to the ability of a system to recover from a failure, for example, whether it withstands heavy traffic, whether it ensures consistency, and whether it recovers after an error, etc. Steel can easily be bent, but a leather belt can recover because of its elasticity. It is the same for applications. Istio not only helps us manage traffic but also provides several failure recovery features that can be dynamically configured at runtime. These features ensure that the service mesh can tolerate failure and prevent it from spreading. This section will introduce how to implement circuit breaking, timeouts, and retries with Istio.

4.4.1 Circuit Breaking

If a service provider does not respond requests when an exception occurs, and the service caller sends a large number of requests without knowing it, it is likely to exhaust network resources, thus bringing down the service caller itself and causing cascading failures. The circuit breaking is designed to prevent such disaster, and the principle is simple: the circuit breaking monitors the status of the service provider, and once an exception is detected and the number of errors reaches a threshold, the circuit breaking will be triggered, and new incoming requests will be blocked and

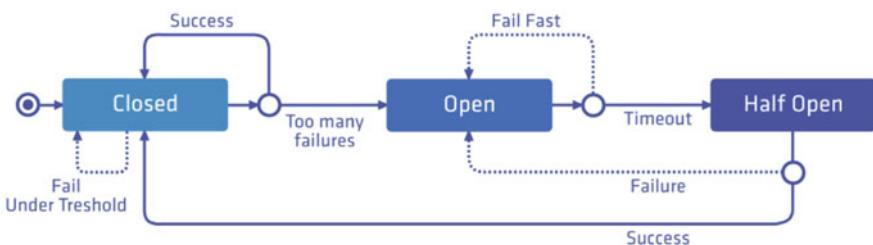


Fig. 4.13 Circuit breaking workflow (from <https://banzaicloud.com/docs/backyards/traffic-management/circuit-breaking/>)

returned with errors, and then continue to be released after a period of time when the service returns to normal, as shown in Fig. 4.13.

Circuit breaking is a bit similar to timeouts in that they are both strategies that trigger a quick failure after a failure occurs, with the advantage of circuit breaking that they also provide an automatic recovery mechanism. This is also true for circuit breaking in other areas, such as stock market, flight, and electrical gate. Designing a circuit breaking starts by focusing on the three states.

- Closed: The breaker is not in effect, and the request can be sent to the service provider normally.
- Open: Service exceptions are detected, and the circuit breaker opens, blocking access to the request.
- Half-open: Put some requests over to see if the service has been recovered.

In addition to the states, two metrics need to be implemented.

- Error threshold: The condition under which the circuit breaking is triggered, such as the number of consecutive errors in invoking the service.
- Timeout clock: The interval to change from open state to half open state.

Istio provides circuit breaker, and we only need to add declarative configuration for the service that needs to be controlled. In Sect. 4.3, we briefly mentioned that circuit breaking needs to be set in the DestinationRule's TrafficPolicy. There are two related configuration items in this object: ConnectionPoolSettings, i.e., connection pool settings for upstream services, which can be applied to TCP and HTTP services. OutlierDetection, or exception detection, is used to track the state of each instance (Pod in Kubernetes) in the upstream service and applies to both HTTP and TCP services. For HTTP services, instances that return 5xx errors in a row will be evicted from the load-balancing pool for a period of time. For TCP services, connection timeouts or connection failures will be considered errors. The specific configuration items for OutlierDetection are shown in Table 4.2.

ConnectionPoolSettings is a configuration item for connection pool management, which is divided into two types: TCPSettings and HTTPSettings.

TCP settings mainly include the maximum number of connections (maxConnections), connection timeout (connectTimeout), long connections (tcpKeepalive), etc. HTTP settings mainly include the maximum number of requests per connection (maxRequestsPerConnection), the maximum number of requests (http2MaxRequests), the maximum number of requests waiting (http1MaxPendingRequests), etc. It is important to note that the connection pool settings can have an impact on the triggering of the circuit breaking; for example, if the number of connections is set to a very small value, the error threshold for the circuit breaking can be easily triggered. The consequences of the configuration need to be fully understood at the time of use.

Let's use a concrete example to demonstrate how to use Istio's circuit breaking. Here is the configuration code for the circuit breaking section in DestinationRule.

Table 4.2 Outlier detection configuration items

Field	Description
consecutiveGatewayErrors	Number of gateway errors before a host is ejected from the connection pool. When the upstream host is accessed over HTTP, a 502, 503, or 504 return code qualifies as a gateway error. When the upstream host is accessed over an opaque TCP connection, connect timeouts and connection error/failure events qualify as a gateway error. This feature is disabled by default or when set to the value 0.
consecutive5xxErrors	Number of 5xx errors before a host is ejected from the connection pool. When the upstream host is accessed over an opaque TCP connection, connect timeouts, connection error/failure and request failure events qualify as a 5xx error. This feature defaults to 5 but can be disabled by setting the value to 0.
Interval	Time interval between ejection sweep analysis. Format: 1 h/1 m/1 s/1 ms. MUST BE > = 1 ms. Default is 10 s.
baseEjectionTime	Minimum ejection duration. A host will remain ejected for a period equal to the product of minimum ejection duration and the number of times the host has been ejected. This technique allows the system to automatically increase the ejection period for unhealthy upstream servers. Format: 1 h/1 m/1 s/1 ms. MUST BE > = 1 ms. Default is 30 s.
maxEjectionPercent	Maximum % of hosts in the load balancing pool for the upstream service that can be ejected. Default is 10%.
minHealthPercent	Outlier detection will be enabled as long as the associated load balancing pool has at least min_health_percent hosts in healthy mode. When the percentage of healthy hosts in the load-balancing pool drops below this threshold, outlier detection will be disabled and the proxy will load balance across all hosts in the pool (healthy and unhealthy). The threshold can be disabled by setting it to 0%. The default is 0% as it's not typically applicable in k8s environments with few pods per service.

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: dr-order
spec:
  host: order
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 5
        maxRequestsPerConnection: 5
    outlierDetection:
      consecutive5xxErrors: 10
      interval: 30s
      baseEjectionTime: 3m
      maxEjectionPercent: 100

```

The error threshold set in this configuration code is 10, which means that 10 consecutive 5xx errors will trigger circuit breaking. In order to make it happen easily, we intentionally set a very small connection pool, and the production environment should be modified as appropriate for its own situation. After the configuration is updated and effective, you can use some load testing tools to test the implementation. Here we use the officially recommended tool fortio and enter the following command.

```
$ fortio load -c 5 -qps 0 -n 100 http://order:9080/1
```

The output is shown below. As you can see from the output, there are a large number of 503 errors when using 5 concurrent.

```
Starting at max qps with 5 thread(s) [gomax 6] for exactly 100 calls (20 per thread + 0)20
:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1 503)
503)20:32:30 W http_client.go:679> Parsed non ok code 503 (HTTP/1.1
503)20:32:30 W http_client.
go:679> Parsed non ok code 503 (HTTP/1.1 503)
...
Code
200 : 11 (4.7 %)
Code 503 : 19 (95.3 %)
Response Header Sizes :
count
100 avg 230.07 +/- 0.2551 min 230 max 231 sum
23007Response Body/Total Sizes : count 100 avg
852.07 +/- 0.2551 min 852 max 853 sum
85207All done 100 calls (plus 0 warmup) 5.988 ms avg, 331.7 qps
```

In Envoy proxy, the indicator of circuit breaking is upstream_rq_pending_overflow, we can enter the container where the proxy is located, find this indicator through pilot-proxy command, and determine whether these 503 errors are caused by circuit breaking, the command is as follows.

```
$ kubectl exec "$FORTIO_POD" -c istio-proxy -- pilot-proxy request GET
stats | grep order | grep pending
```

The output results are as follows.

```
cluster.outbound[9080] | order.app.svc.cluster.local.upstream_rq_pending_overflow:
95cluster.outbound[9080] | order.app.svc.cluster.local.upstream_rq_pending_total: 100
```

As you can see from the output, there is upstream_rq_pending_overflow with a value of 95, which matches the number of 503 errors.

Circuit breaking helps reduce the resources occupied in potentially failed operations to avoid waiting timeouts on the calling side. However, Istio's circuit breaking is not perfect compared to Hystrix and cannot define a degradation operation after a circuit breaking occurs as the latter does. Therefore, you need to consider what you should do after a circuit breaking occurs or implement the corresponding fallback logic yourself when using it.

4.4.2 *Timeouts and Retries*

In this section, we will describe how to set the timeout and retry and what are the considerations when using them.

Timeout

In principle, a timeout mechanism should be set for any remote call, especially one that spans multiple processes (even if they are on the same host). The implementation of the timeout is not difficult; the hard part is how much timeout is set to be appropriate. A timeout that is too long reduces its effectiveness and may cause delays in waiting for a service response, and client resources are consumed while waiting. And too short a timeout can lead to unnecessary failures and increase the load on the back-end if it is retried too many times. The author's experience is to use the latency indicator of the service as the reference value. First, we need to count the latency of the service, which can be analyzed by request logs; then choose an acceptable timeout ratio, such as 1%, accordingly, the timeout we set should be the latency time of P99. Of course, this approach is not applicable to all cases, such as not being suitable for the case of severe network latency, or the case of infrastructure service problems. There is also a situation where the difference in latency is very small; for example, P99 and P50 are very close to each other, at which time needs to be increased appropriately.

In Sect. 4.3, we mentioned that Istio's timeout is very simple to set; just configure a timeout field in the virtual service. The field is in seconds (s) and can be set to a decimal number, e.g. 0.5 s. Note also that the timeout is only supported for `HTTPRoute` configuration items. By default timeout is turned off; i.e., if you want to add a timeout feature to a service, you need to configure it manually. The following example shows the process of setting a 2 s timeout for an order service.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: vs-timeouts
spec:
  hosts:
    - order
  http:
    - route:
        - destination:
            host: order
        timeout: 2s
```

Then we can call the order service by curl command, and from the output logs we can see that the request is hijacked by Envoy proxy, and the delay is indeed about 2 s.

```
$ k exec -it demo-557747455f-jhhwt -c demo -- curl -i order:  
9080/1HTTP/1.1 200  
OKcontent-type: text/html;  
charset=utf-8content-length:  
1683server :  
envoydate: Tue, 18 May 2021 04:04:23  
GMTx-envoy-upstream-service-time: 2069 (ms)
```

You can also view the timeout metrics with the pilot-proxy command as follows.

```
$ kubectl exec "SORDER POD" -c istio-proxy -- pilot-proxy request GET stats | grep  
timeoutcluster.order.upstream_rq_connect_timeout: Ocluster  
.order.  
upstream_rq_per_try_timeout: Ocluster.  
order.upstream_rq_timeout: 1  
...
```

Retry

Retries are a way to improve system availability when network jitter or random errors are encountered, but they still need to be used with care. Client-side retries mean that it will take more server time to get a higher success rate. This is not a problem in the case of occasional failures since the total number of retry requests is small. However, if the failure is caused by overload, retries will instead increase the load and cause the situation to deteriorate further. Based on the author's experience, retry needs to be designed and used with the following points in mind.

- Set the number of retries: The meaning behind retries is that we think the failure is temporary and can be recovered quickly so that retries make sense. Therefore, setting a retry limit is necessary to avoid generating unnecessary loads.
- Fallback strategy: Fallback is the preferred solution to reduce the negative effects of retrying. Instead of retrying immediately, the client will wait for a period of time between attempts. The most common strategy is exponential fallback, or exponential fallback, where the waiting time after each attempt is extended exponentially, e.g., the first retry is after 1 s, the second is after 10 s, and so on. Exponential fallback may lead to long fallback times, so it is usually used in conjunction with the number of retries.
- Idempotency restriction: Retries are only safe if the API is idempotent. Read-only APIs like HTTP GET is idempotent, but not necessarily for add-delete-change APIs. So you must know whether the API is idempotent before using retry.
- Beware of multi-layer retries: The call chain of a business transaction in a distributed system is likely to be complex. If a retry policy is set for each layer of invocation, it may result in the retry mechanism not working properly at all when a failure occurs. For example, if a business behavior cascades to invoke 5 different services, assuming 3 retries at each layer, it will retry 9 times at the second layer, 27 times at the third layer, and so on. Therefore a reasonable solution is to set up retries for a single node in a chain of calls.

Table 4.3 Retry configuration items

Field	Description
Attempts	Number of retries to be allowed for a given request. The interval between retries will be determined automatically (25 ms+). When request timeout of the HTTP route or per_try_timeout is configured, the actual number of retries attempted also depends on the specified request timeout and per_try_timeout values.
perTryTimeout	Timeout per attempt for a given request, including the initial call and any retries. Format: 1 h/1 m/1 s/1 ms. MUST BE \geq 1 ms. Default is the same value as the request timeout of the HTTP route, which means no timeout.
retryOn	Specifies the conditions under which retry takes place. One or more policies can be specified using a ',' delimited list. If retry_on specifies a valid HTTP status, it will be added to retriable_status_codes retry policy. See the retry policies and gRPC retry policies for more details.
retryRemoteLocalities	Flag to specify whether the retries should retry to other localities. See the retry plugin configuration for more details.

- Load: Retries inevitably lead to an increase in load as the number of requests increases. Circuit breaking is the main solution to this problem; i.e., requests sent to the service provider are completely blocked by fast failures.
- Determine retry scenarios: You need to determine which errors require to retry and which don't; which APIs are important and need retry to improve availability, and which APIs won't have much impact even if there is a problem. Not all failure scenarios need retries, and setting retries requires a reasonable tradeoff.

Retries can be implemented in Istio through the `HTTPRetry` configuration item in the virtual service, with the specific fields shown in Table 4.3.

The following example shows the process of setting up a retry for the order service, triggered when the system has an error such as 5xx. The service's logs can be viewed to determine if a retry is in effect. Likewise, the number of retries can be determined by looking at the retry metric `upstream_rq_retry`.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name:
name: vs-retry
spec:
hosts:
- order
http:
- route:
- destination:
  host: order
retries:
attempts: 3
```

```
perTryTimeout: 2s
retryOn: 5xx,gateway-error,reset
```

Despite some limitations in its use, retry is still an important resiliency mechanism to improve system availability. One thing to remember is that retries are a judgment call by the client based on its own situation and a way to get the server to spend more resources to handle the request. If the client uses retries uncontrollably, it is likely to cause more problems and therefore needs to be used with caution.

4.5 Summary

In this chapter, we focus on how to manage the traffic of microservice applications through the capabilities provided by Istio. In Sect. 4.1, we introduce two types of traffic: north-south traffic and east-west traffic, which represent the traffic coming into the application from outside and the traffic between services inside the application. Also, we propose the service mesh as the preferred traffic management solution in the cloud-native era. In Sect. 4.2, we analyze the changes in the service discovery mechanism after migration to the cloud. The last two sections detail the core customization resources of Istio Service mesh and how it can be used to implement traffic management such as routing, circuit breaking, timeout, and retry.

In terms of architectural evolution, Service mesh, a transparent, declarative configuration approach to traffic management, has become an important infrastructure in architecture. For building cloud-native applications, the author recommends it as the primary technology selection option.

Chapter 5

Distributed Transactions



As software systems move from monolithic applications to microservices and cloud-native services, and with the trend of decentralized and heterogeneous database selection, can ACID transactions, previously built on monolithic applications and single traditional relational databases, achieve the same functionality on distributed systems and diverse databases? If so, what pitfalls and considerations are essential in the transitional mindset? This chapter will introduce our team's experience and practice pertaining to the technical topic of distributed transactions.

5.1 Theoretical Foundations

This section introduces the background of distributed transactions and related concepts, including ACID, CAP, BASE, etc., starting from the perspective of architecture and business changes.

5.1.1 *Background*

As the business and architecture are constantly changing, software systems tend to fall into a seemingly inevitable situation, where the business model and the data storage model cannot be aligned. This is the reason why transactions came into being. While traditional ACID transactions were widely studied and commonly used in the era of monolithic applications, distributed transactions are an increasingly prominent engineering problem in the context of microservices.

1. Business Changes

Software systems, in order to achieve certain business functions, will abstractly represent people, things, and objects in the real world and map them

into models in software systems. Borrowing ideas from systems theory, a generic business model can be constructed in the following steps.

- Define what entities exist in the system and what attributes are available on the entities. For example, for a trading system, buyers, sellers, orders, and traded goods are all entities, and for the entity of orders, there may be attributes such as transaction time and transaction price.
- Define various topological relationships between entities, such as subordinate, nested, and many-to-many. For example, buyers and sellers are many-to-many relationships, and transacted goods and orders are subordinate to each other.
- Define the dynamic relationships of entities and attributes, which are the processes of the system. In order to record and track changes in the system and make the system's behavior easy to understand and review, the process is usually abstracted as changes in several key attributes on a set of entities; these changes can be subdivided into states (the values of key attributes, usually finite and discrete values), transfers (which states can change to which states), and conditions (what constraints are satisfied to achieve the transfer of states). For example, a transaction in the system, from placing an order to confirming receipt, is a process that is driven by a change in the state of the order as an entity, and the state transfer of the order needs to satisfy some conditions, such as sufficient inventory.

The first two steps can be captured by drawing the entity-relationship diagram of the system, which represents all possible states of the system at any one time point. The third step can be expressed by a flowchart and a state-transfer diagram, which represent the system's change pattern over time (or sequence of events).

In particular, when designing the state-transfer diagram of a system, it usually demands that the values of the state are mutually-exclusive and collectively inclusive, i.e., the system is in a given state at any point in time and cannot be in two states at the same time. Only then can the state-transfer diagram completely and correctly characterize all possible states and change patterns of the system.

But as stated in the previous sections, the single constant in a system is the change itself, and the business model is usually the most change-prone part. Corresponding to the steps of business modeling, changes in the business model are usually reflected in the following:

- Increase or decrease in the types of entities and attributes in the system, such as the subdivision of the buyer entity into individuals and merchants, and the addition of attributes such as billing method and shipping address to the order entity.
- Changes in entity topology, such as orders, can be split into sub-orders and packaged orders.
- Process changes, such as the need to add the step of locking inventory before placing an order.

In addition, even if the entities and relationships in the system remain the same, the number of certain types of entities and relationships may change, such as the number of tradable items suddenly increasing to 100 times the original number, or the number of new orders added daily suddenly increasing to 10 times the original number.

2. Architectural Changes

Although the business value of software systems lies in the ability to achieve certain functions, the prerequisite of completing these functions is to meet some basic requirements unrelated to the specific business, such as stability, scalability, parallel development efficiency, energy-utility ratio, and so on. Along with the surge in data volume and functional complexity, the technical architecture and engineering methodology supporting R&D are evolving and innovating.

Software systems follow the computation-storage architecture proposed by computer science pioneer John von Neumann, which stands that most of the components are stateless or volatile computational units, and only a few components are responsible for “persisting” data onto a disk. The components responsible for storing these components are the database, and the other computational components that read and write data can be regarded as the clients of the database. The design of the data storage model is basically equivalent to the database design, which includes data storage structure, index structure, read and write control, etc.

In the traditional monolithic application era, software systems usually use a single relational database (RDBMS): a software system corresponds to a database, an entity corresponds to a table, and the attributes on the entity correspond to the fields in the table, usually to ensure the consistency of the business model data, but also to minimize redundant storage tables and fields, forming the so-called database design paradigm.

In the context of microservices, the entire software system is no longer limited to using a single database; in order to meet the needs of diverse indexing and query data, each service can even choose to use a number of special databases specializing in certain areas, such as Apache Solr and ElasticSearch as the representative of the search engine (Search Engine). The database can be distributed to access large amounts of data, such as Amazon DynamoDB and MongoDB, which are NoSQL document databases.

3. Transactions

On the one hand, business changes drive the iterative business model, which inevitably leads to changes in the data storage model to some extent, that is, the addition and deletion of database tables, fields, indexes, and constraints, as well as changes in the mapping relationship from the business model to the storage model; on the other hand, non-functional requirements continue to drive the evolution of technical architecture and engineering methodology. On the other hand, non-functional requirements continue to drive the evolution of technical architectures and engineering methodologies, making database selection more diverse and decentralized.

In order to accommodate these two changes, it seems unavoidable that the state of the business model and the state of the storage model diverge and do not correspond one-to-one in the process of continuous service boundary adjustment. For example, after a buyer makes a payment, a series of changes need to be completed such as deduction of product inventory, increase or decrease of buyer and seller account balances, modification of order status to paid, etc. That is, the transfer of the business model from the state of pending payment to the state of paid needs to correspond to the joint changes of multiple fields on the inventory table, order table, and even more related tables. For example, in order to improve the conversion rate of users from browsing to purchasing, the previous multi-step order placement process is changed to one-step creation, which is equivalent to the reduction and merging of the number of states on the business model, while the data storage model must be adjusted accordingly.

In order to ensure the mutually exclusive completeness of the state on the business model, the software system inevitably requires that the state of the storage model is also mutually exclusive: from the perspective of the database, some fields on some tables in the database must change at the same time; and from the perspective of the database client, a set of write operations to the database either take effect or they do not. This constraint is actually a transaction.

5.1.2 ACID: Transaction Constraints in the Traditional Sense

In a traditional relational database, a set of data operations that succeed and fail at the same time is called a transaction and consists of four aspects, abbreviated as ACID.

- A (Atomicity, Atomicity): A set of data operations; if one of the steps of the operation fails, the previous operations should also be rolled back, not allowing the case of partial success and partial failure.
- C (Consistency, Consistency): Data operations conform to some business constraints. This concept comes from the field of financial reconciliation, and the meaning of expanding to database design is rather vague, and there are many different opinions. Some sources even say that C is added to make up the acronym ACID.
- I (Isolation): There is a certain amount of isolation for concurrent data operations. The worst case is that concurrent operations are not isolated from each other and interfere with each other; the best case is that concurrent operations are equivalent to a series of serial operations (Serializable). The higher the isolation level, the more resources the database needs, and the worse the performance of accessing data (e.g., throughput, latency).
- D (Durability): Requests arriving at the database will not be “easily” lost. Usually, database design documents will define “easily” specifically, for

example, not losing data during disk bad physical sectors, machine power outages, and restarts.

5.1.3 CAP: The Challenge of Distributed Systems

As the topology of software systems moves from monolithic applications into the era of microservices and the number and variety of databases grow, distributed systems, especially databases, face greater challenges than monolithic applications and traditional relational databases in meeting the transactional requirements of traditional ACID standards.

The so-called CAP triple-choice theorem states that no distributed system can satisfy all three of the following properties at the same time.

- C (Consistency, strong consistency): Any node of the distributed system for the same key read and write requests, the results obtained exactly the same. Also called linear consistency.
- A (Availability): Each request gets a timely and normal response, but there is no guarantee that the data is up-to-date.
- P (Partition tolerance): The distributed system can maintain operation if the nodes cannot connect to each other or if the connection times out.

Among the three characteristics of CAP, tolerating network separation is usually a given fact that cannot be avoided: if there is no network separation in the system, or if the system is not required to work properly in case of network separation, then there is no need to adopt a distributed architecture at all, and a centralized architecture is simpler and more efficient. With the acceptance of P, the designer can only make a trade-off between C and A. Few distributed systems insist on C.

Few distributed systems insist on C and abandon A, i.e., choose strong consistency and low availability. In such systems, a request to write data returns a response only after it has been submitted and synchronized to all database nodes, and a failure of any node or network separation will result in the service being unavailable as a whole until the node failure is recovered and the network is connected. The service availability depends on the frequency of failure and recovery time. This choice is usually made for systems involved in the financial sector, even without a distributed architecture.

Most systems, on balance, choose A and reduce the requirements for C, with the goal of high availability and eventual consistency. In such systems, a request to write data returns a response as soon as it is successfully submitted on some of the database nodes, without waiting for the data to be synchronized to all nodes. The advantage of this is that the availability of the service is greatly increased, and the system is available at any time as long as a few nodes are alive. The disadvantage is that there is a certain probability that the result of reading the same data will be wrong for a period of time (unknown length and no upper limit) after the write data request is completed. This data constraint is called BASE.

5.1.4 *BASE: The Cost of High Availability*

BASE is a weaker transactional constraint than ACID in distributed systems, and its full name is **B**asically **A**vailable, **S**oft state, **E**ventually consistent (eventually consistent). Here is a look at the meaning of each of these terms.

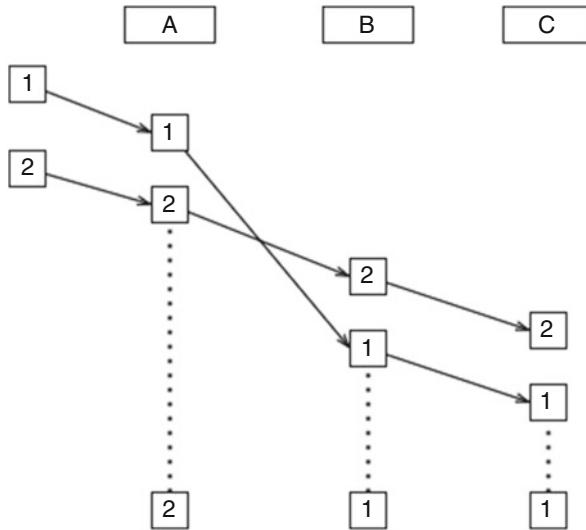
- Basic Availability: By using a distributed database, read and write operations are kept in as usable a state as possible, but data consistency is not guaranteed. For example, a write operation may not be persistent, or a read operation may not return the most recently written value.
- “Soft” state: The state of a piece of data is unknown for some time after it is written, and there is only a certain probability that the system will return the latest written value before it finally converges.
- Eventually consistent: Under the premise that the system functions normally, after waiting long enough, the state of a piece of data in the system can converge to a consistent state, after which all read operations on it will return the latest written value.

BASE is a detailed representation of the decision to choose high availability and relax data consistency in the design of distributed systems. It discusses the state changes that occur in a data system after data is written: the state of the data on multiple slices goes from consistent to inconsistent to consistent, and the value returned by a read operation goes from stable to the correct old value, to unstable to both old and new values, to stable to the new value.

It should be noted that BASE mainly discusses the behavior of data read and write operations in a data system; another situation often occurs in practical applications, that is, a data stored in multiple data systems. Let’s look at an example of data consistency across multiple data systems.

5.1.5 *Write Order*

A distributed system with a distributed or even heterogeneous data storage scheme may have data inconsistencies caused by concurrent write order differences on different services/databases for the same piece of data. For example, there are three services, A, B, and C, each using a different database, and now there are two requests, 1 and 2, concurrently modifying the same data item X. Different fields of X are processed by A, B, and C, respectively. Due to random network delays, X ends up landing in the three services/databases with inconsistent values, A with a value of 2 and B and C with a value of 1. It looks like the Fig. 5.1 as below.



Inconsistent data due to write order differences

This multi-database write data order inconsistency problem does not necessarily violate the BASE or ACID constraints, but the system is also in an unintended state from a business model perspective.

5.2 Solution Selection for Distributed Transaction Framework

Based on the background of distributed transactions and related concepts, this section will discuss some topics of distributed transaction solution selection. First, we will introduce some relevant practices in academia and industry, and then we will introduce a distributed transaction solution developed by the FreeWheel core business team, including its design goals, technical decisions, overall architecture and business processes, etc.

5.2.1 Existing Research and Practice

As a technical challenge and a business necessity, there are a lot of research and practice on distributed transactions in both academia and industry, listed below, to name a few.

1. XA Standard and Two-Phase Commit Protocol

The XA standard is an abbreviation for eXtended Architecture (the extended architecture here is actually ACID transactions), which is led by The Open Group to try to provide a set of standards for distributed transaction processing. XA describes the interface between the global transaction manager and the local resource manager. Through this interface, applications can access multiple resources (e.g., databases, application servers, message queues, etc.) across multiple services within the same transaction that maintains ACID constraints. XA uses the Two-phase Commit (abbreviated as 2PC) protocol to ensure that all resources commit or roll back any particular transaction at the same time.

The two-phase commit protocol introduces the role of an orchestrator, which is responsible for unifying the results of operations across data storage nodes (called participants). In the first phase, participants perform data operations concurrently and notify the coordinator of their success; in the second phase, the coordinator decides whether to confirm the commit or abort the operation based on feedback from all participants and communicates this decision to all participants.

The advantages of using XA and two-phase commit to implement distributed transactions are

- Strong consistency: ACID constraints on data across multiple databases are achieved.
- Less business intrusive: Distributed transactions rely entirely on the support of the individual databases themselves and do not require changes to the business logic.

The disadvantages of using XA to implement distributed transactions are also obvious.

- Database selection restrictions: The database selection for the service introduces the restriction of supporting the XA protocol.
- Low availability and network fault tolerance: The coordinator or any one of the participants is a single point of failure, and no network separation can occur between any components.
- Low performance: Databases that support XA features are designed with a large number of blocking and resource-occupying operations and have poor data volume and throughput scalability.

XA is designed to be a strongly consistent, low-availability design solution that does not tolerate network separation, and although it meets the constraints of ACID transactions, its practice in industry is quite limited and usually confined to the traditional financial industry.

2. Saga

Saga originally means a long mythical story. It implements distributed transactions with the help of a driven process mechanism that executes each data operation step sequentially and, in case of failure, performs the “compensating” operations corresponding to the previous steps in reverse order. This requires that the services involved in each step provide a compensating operation interface that corresponds to the forward operation interface.

The advantages of using Saga to implement distributed transactions are as follows:

- Microservices architecture: A number of underlying services are combined/organized to fulfill various business requirements.
- High database compatibility: There is no requirement for each service to use any database technology, and the service can even be database free.

The disadvantages of using Saga to implement distributed transactions are as follows:

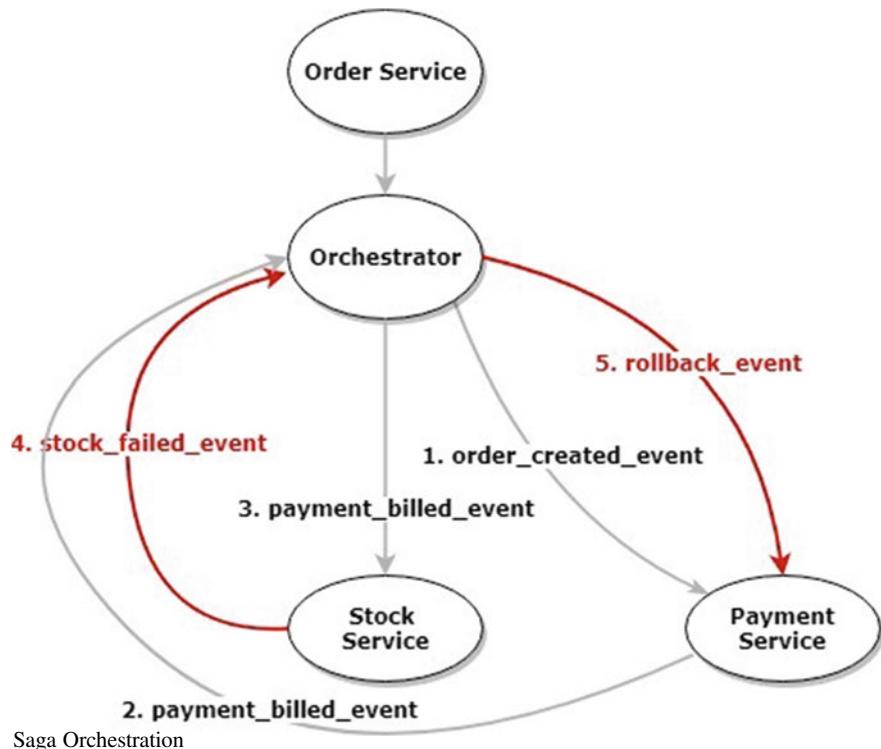
- Requires the service to provide a compensating interface: increases the cost of development and maintenance.
- Not ACID compliant: Isolation (I) and durability (D) are not addressed.

Saga can also be divided, process-wise, into two variants: Orchestration (symphony) and Choreography (dance in unison).

- Saga Orchestration

Saga Orchestration introduces a role similar to that of the coordinator in XA to drive the entire process.

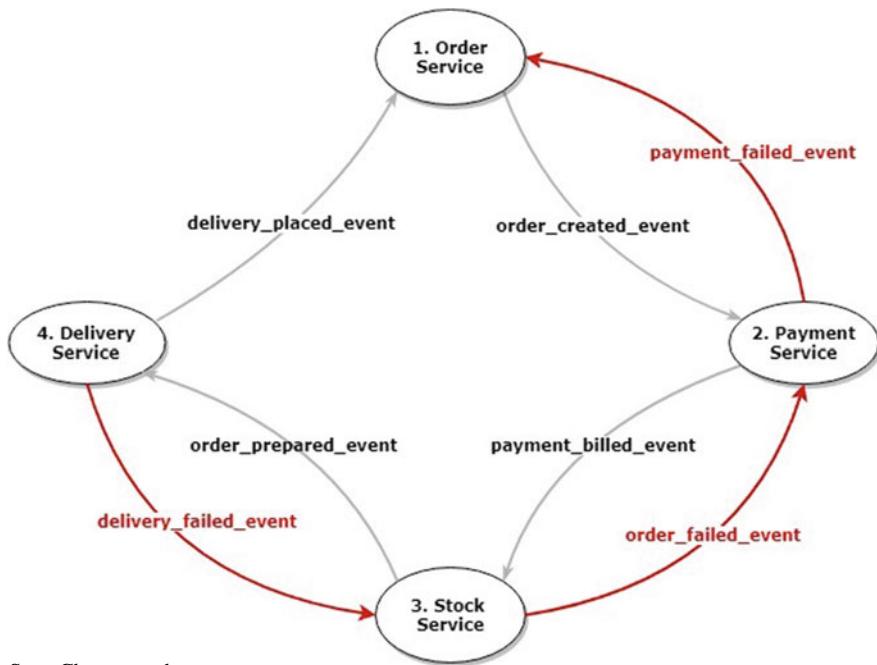
As shown in the diagram, Order Service initiates the distributed transactions, Orchestrator drives the distributed transaction process, and Payment Service and Stock Service provide the forward and compensation interfaces for data manipulation.



- **Saga Choreography**

Saga Choreography, on the other hand, splits the process into the services involved in each step, with each service calling the back-order or front-order service itself.

As shown in the diagram, the Order Service directly calls the Payment Service to initiate a distributed transaction, which in turn calls the Stock Service until all steps are completed; if a step fails, the call is reversed between the services.



Saga Choreography

3. ACID Transaction Chains

ACID transaction chaining can be seen as a Saga Choreography enhancement that requires all services involved in a distributed transaction to use a database that supports traditional ACID transactions, and within each service, packages data operations and synchronous invocations of neighboring services into an ACID transaction, enabling distributed transactions through chained invocations of ACID transactions.

The advantages of using ACID transaction chaining to implement distributed transactions are as follows:

- ACID-compliant: Each step is a traditional ACID transaction, and the whole is ACID-transactional.
- No need for a service to provide a compensation interface: Rollback operations are performed by a database that supports ACID transactions.

The disadvantages of using ACID transaction chains for distributed transactions are as follows:

- Database selection restrictions: The database selection for the service introduces the restriction of supporting traditional ACID transactions.
- Too much service coupling: The dependency between services is a chain topology, which is not convenient to adjust the order of steps; as the number of various business processes using distributed transactions increases, it is easy

to generate circular dependencies between services, causing difficulties for deployment.

5.2.2 *Design Goals of Distributed Transaction Framework*

After investigating the above industry practices, the FreeWheel core business system decided to pursue developing a distributed transaction solution in order to achieve the following design goals:

- The business can define a set of data operations, i.e., distributed transactions, which either succeed or fail at the same time, regardless of the service and database in which they occur. Whenever any operation in the transaction fails, all previous operations need to be rolled back.
- Overall high availability of the system: When some nodes of some services fail, the system as a whole is still available. By supporting the rapid expansion and shrinkage of services to achieve high throughput of the system as a whole, as short as possible, to achieve consistency of data latency. The framework itself consumes low resources and introduces little latency overhead.
- Data eventual consistency: Concurrent operation of the same data requests arrives at each service and database in the same order, without the previously mentioned inconsistent write order phenomenon.
- Support for service-independent evolution and deployment: There are no requirements or assumptions about how services are implemented, except for support for communication using RPC and a given protocol.
- Support services to use heterogeneous data storage technologies: Using different data storage technologies (relational databases, NoSQL, search engines, etc.) is the status and effort of each service of FreeWheel's core business system.
- The architecture is less invasive and easy to adopt: No or less changes are made to the code and deployment of the existing system, and as much as possible, the operating environment and specific business processes of distributed transactions are implemented only by adding new code as well as service deployment. Clear division of labor between framework and business, maintaining 100% test coverage for framework code, 100% testability for business code, and low testing costs. Maintain high visibility and predictability of the system, as far as possible, to facilitate rapid fault location and recovery.
- Support for synchronous and asynchronous processes: Provide a mechanism to bridge the synchronous interaction process between the UI/API and the back-end entry service, with the possible asynchronous process between the back-end services.
- Support for transaction step dependencies: Whether and how data operations at a step inside a transaction are executed depends on the results of the operations of the preceding steps.

5.2.3 Choosing Saga

We first rule out the XA solution, which does not meet the high availability and scalability of the system. Second, we ruled out the ACID transaction chain because it is not compatible with the existing database selection of the business, and more database technologies that do not support ACID transactions will be introduced in the future.

The final decision to use Saga to implement a highly available, low-latency, and eventually consistent distributed transaction framework was based on the fact that its design ideology fits well with the current SOA/microservices/Serverless practice of the FreeWheel core business team, i.e., by combining/orchestrating some basic services (actually Lambda for Serverless, not to be distinguished here) to the combination/orchestration of some basic services (actually Lambda for Serverless) to fulfill various business requirements.

After comparing the two variants of Saga, we chose Orchestration over Choreography for the following reasons:

- Service decoupling: Orchestration naturally decouples the driver logic of the transaction itself from the many underlying services, while Choreography is prone to circular dependency problems between services without introducing queues.
- Service layering: Orchestration naturally separates services into two invocation levels, the combination/orchestrator and the domain service, which facilitates the extension and reuse of business logic.
- Data decoupling: For business scenarios where a step depends on the results of multiple steps in the previous order, the latter requires all services in the previous order to pass through data from other services, which Orchestration does not.

By adopting Saga Orchestration, two of its drawbacks must be overcome, namely, the requirement for the underlying service to provide a compensation interface and the lack of implementation of the isolation and durability constraints in ACID.

1. Implementing data compensation operations

Data operations can be divided into Insert (New), Delete (Delete), and Update (Update) three, and Update can be subdivided into Full update (Replace, overall update) and Partial update (Patch, partial update); they correspond to the following compensation operations.

- Insert: The compensation operation is Delete, the parameter is the ID of the data, and the ID of the data is required to be recorded after the Insert operation.
- Delete: The compensating operation is Insert with the parameter of complete data, which requires the current complete data to be recorded before the Delete operation is performed.

- Full update: The compensating operation is another Full update with the parameter Full data, which requires the current full data to be written down before the original Full update operation.
- Partial update: The compensation operation is a Partial/Full update with the parameter Partial or Full data before the change, which requires the current partial or full data to be written down before the original Partial update operation.

2. Implementing Isolation and Durability Constraints in ACID

Isolation is really a question of how to control concurrency, i.e., how to handle concurrent operations on the same data (same key). MySQL, as one of the mature relational databases, has introduced the Multiple Version Concurrency Control (MVCC) mechanism. The main idea to control concurrency without introducing multiple versions is to remove concurrency and turn it into a string, which has two main types of implementations: preemptive locks or using queues. Considering the performance loss due to waiting for locks and the possibility of interlocking due to inconsistent order of multiple locks, we prioritize the use of queues to remove concurrency.

Durability means that a transaction successfully committed to the system cannot be lost in the middle, i.e., data persistence is achieved. The failures to be considered include the failure of data storage nodes and the failure of data processing nodes.

In summary, in order to comply with the ACID constraint, a queue + persistence technology solution is needed to complement the two shortcomings of Saga. Combined with the existing infrastructure mapping of FreeWheel's core business system, we prioritized the introduction of Apache Kafka (hereinafter referred to as Kafka).

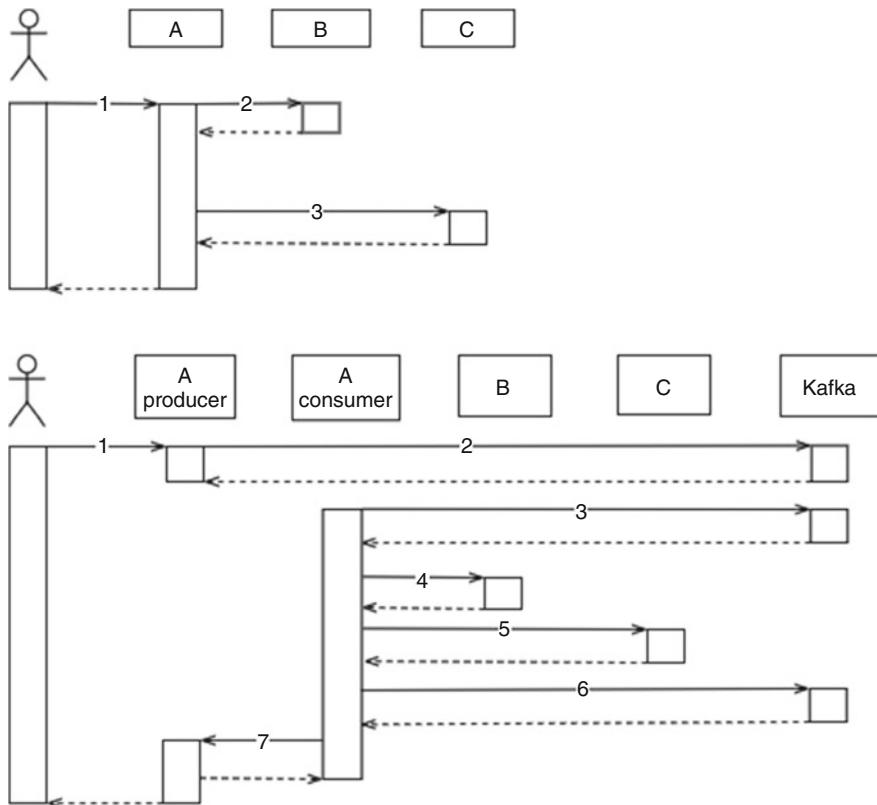
5.2.4 *Introducing Kafka*

Kafka is queue-plus persistence solution with rich features designed for distributed systems, including the following capabilities:

- Message Preservation: Introducing queues to turn concurrent writes into strings and solve the isolation problem of concurrent writes.
- Message Delivery Guarantee: Supports “at least once” message delivery guarantee, with redundant backup and failure recovery capabilities, which helps solve the ACID durability problem.
- Excellent performance: Various sources show that Kafka itself is the industry benchmark for efficiency and reliability, and if used properly, it will not become a performance bottleneck for the system.

On the other hand, as a powerful queuing solution, Kafka brings new challenges to the design and implementation of distributed transactions.

For example, before the introduction of queues, the nodes in the main process were interacting synchronously from the time the customer clicked the browser button to the time the data was dropped and the response data was returned; after the introduction of queues, the producers and consumers at the two ends of the queue are separated from each other, and the whole process switches from synchronous to asynchronous and back to synchronous again, as shown in the figure, where the solid arrows are RPC requests and the dashed arrows are RPC responses, and the data is processed in the order of the steps marked by the serial number. The data is initiated from the client in the order of the steps marked by the serial number and passes through services A, B, and C.



Synchronous-async conversion

As you can see, before the introduction of the queue, all steps are executed in synchronous order; after the introduction of the queue, Steps 1 and 2 are synchronous, 2 and 3 are asynchronous, and the next Steps 3 through 7 are synchronous again.

Although the overall throughput and resource utilization of the system can be further improved by turning synchronous into asynchronous, the design of how to connect synchronous and asynchronous processes needs to be added in order to maintain the synchronous front-end data process.

1. Synchronous-asynchronous conversion mechanism design

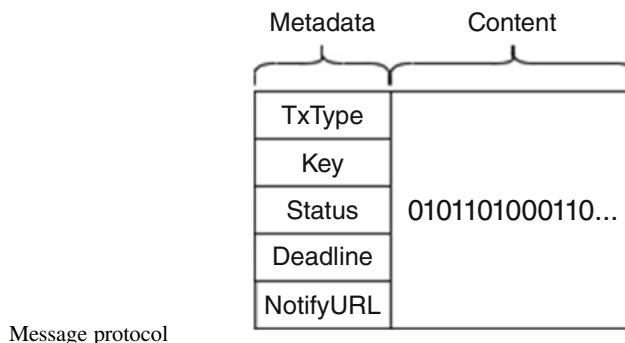
Synchronous to asynchronous conversion is relatively simple and can be achieved by sending messages to Kafka asynchronously through the Goroutine in Go or the thread mechanism in Java, etc., which is not discussed here.

Asynchronous to synchronous is a bit more complicated and requires a mechanism for peer-to-peer communication between the node where the consumer is located and the node where the producer is located. One approach is to introduce another Kafka queue, where the consumer finishes processing a message, encapsulates the result in another message, and sends it to the queue, while the producer's process starts a consumer that listens and will process it.

We take a different approach: the producer wraps the callback address into the message, and the consumer sends the result to the callback address after processing is complete. This works because in our existing deployment scenario, the networks of the nodes where the producer and consumer are located are interoperable.

2. Queue Message Protocol Design

A queued message for a distributed transaction contains at least two parts of information: Metadata and Content. The figure as below shows the content of message.

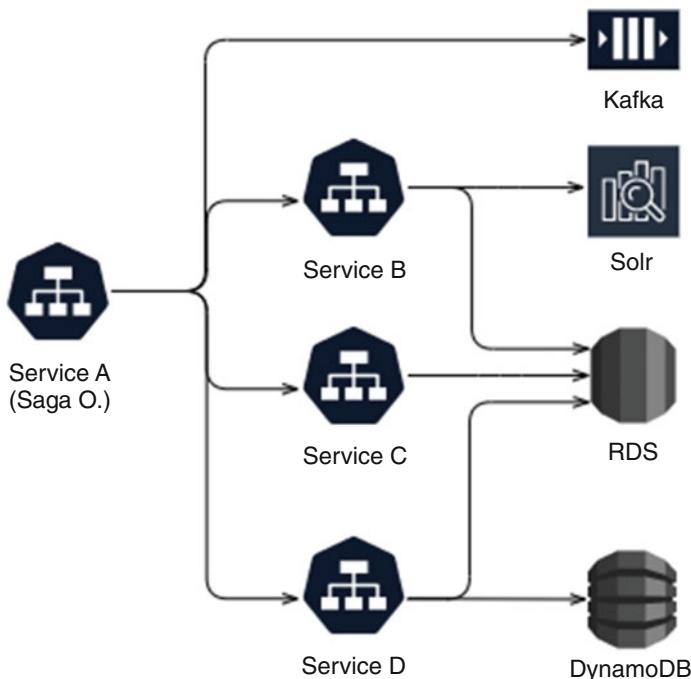


- **Metadata:** Read and written by the distributed transaction framework, using JSON format, field format fixed, and business code can only read, not write. The most important field in the metadata is the type of the distributed transaction message (hereafter referred to as TxType). The producer specifies the TxType of the message by the strong type; the distributed transaction framework in the consumer process performs event sourcing based on the TxType and invokes the corresponding business logic for consumption.

- Content: Read and write by the business code, the format is arbitrary, and the framework does not parse, as long as the length does not exceed the limit of the Kafka topic (default 1MB).

5.2.5 System Architecture

The architecture of the distributed transaction system based on Saga Orchestration and Kafka is shown in the following figure.



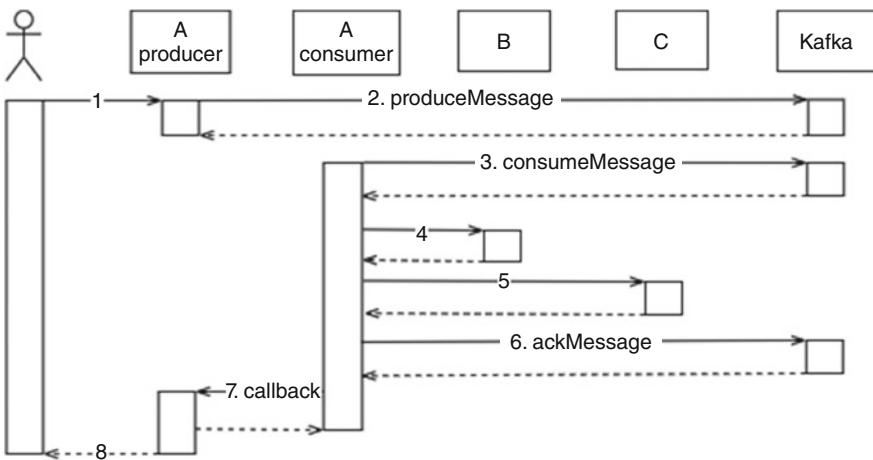
Distributed transaction architecture

Service A is the orchestration organizer that drives Saga Orchestration's processes, and services B, C, and D are the three underlying services that use separate and heterogeneous databases.

Since Saga Orchestration is used instead of Choreography, only service A is aware of the presence of distributed transactions and has dependencies on Kafka and Saga middleware, while the domain services B, C, and D only need to implement a few more compensation interfaces for A to call, without creating dependencies on Kafka and Saga.

5.2.6 Business Process

The flow of service A from receiving a user request, triggering a distributed transaction, invoking each domain service in steps, and finally returning a response is shown in the figure.



Business process

Step details:

- 1 and 2: After receiving a user request, a node of service A first assumes the role of producer, wrapping the user request and callback address into a message and sending it to Kafka, and then the processing unit handling the user request blocks and waits.
- 3–5: A consumer in a node of the same service A receives a message from Kafka and starts driving the Saga Orchestration process, invoking the interfaces of services B and C in the order and logic defined by the business.
- 6 and 7: At the end of the Saga process, the consumer sends an acknowledgment of consumption progress (ackMessage, i.e., updates the consumer group offset) to Kafka and then sends the result (success or failure, what changes were made) to the producer via an RPC callback address.
- 8: After receiving the data from the callback address, the producer finds the corresponding user request processing unit, unblocks it, and finally encapsulates the result into a user response.

5.3 Distributed Transactions Based on Saga and Kafka in Practice

After clarifying the design scheme of the distributed transaction system, this section describes some problems and attempts to solve them during testing and live operation.

5.3.1 *Improvements to Kafka's Parallel Consumption Model*

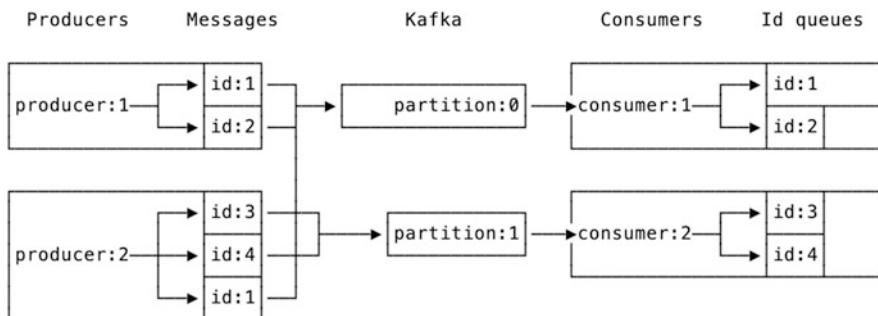
The message data on Kafka is divided into topic and partition hierarchies, with topic, partition, and offset uniquely identifying a message. Partition is the hierarchy responsible for ensuring message order. Kafka also supports multiple consumptions of a message by different “services” (called multicast or fanout), and to distinguish between different “services”, introduces the concept of consumer groups, where a consumer group shares a consumption schedule on a partition (consumer group offset). To ensure the order of message delivery, the data on a partition is available to at most one consumer at the same time and in the same consumer group.

This incurs some practical issues for Kafka users:

- Overestimation of partitions leads to wasted resources: the number of partitions on a given topic can only increase, not decrease, in order not to lose messages. This requires a topic to estimate its production and consumption capacity before going live, then deploy it with an upper limit of production capacity and a lower limit of consumption capacity, and then set an upper limit of the number of partitions. If the production capacity of the topic is found to be higher than the consumption capacity, the partition must be expanded first, and then the consumption capacity must be increased (the most direct way is to increase the number of consumers). On the contrary, if the production capacity on the topic is found to be lower than the consumption capacity (either because the production rate of messages is lower than expected or fluctuates significantly, or because the consumption capacity of individual consumers has been increased through optimization), the number of partitions cannot be scaled back, resulting in a waste of Kafka's resources. The reality is that the number of partitions is often overestimated, and the processing power of the Kafka topic is often wasted. That's why business development engineers design various reuse mechanisms for topics and partitions.
- Partitioning is not sufficient to distinguish between messages that need to be consumed serially and those that can be consumed in parallel: Kafka's default message partitioning strategy is to assign messages to a specific partition by computing hash values for their key fields, but it is possible that a group of consumers may not need to consume all messages on a partition serially. For example, if a service believes that messages A, B, and C are all partitioned into

partition 0, but only A and C have a sequential relationship (e.g., the same data is updated), B can be consumed in parallel with A and C. If there is a mechanism that allows the business to define which messages need to be consumed serially and the rest can be consumed in parallel, it can improve the consumption parallelism and processing power without changing the number of partitions and reduce the dependence of code on the number of partitions.

To address the above two issues, distributed transactions make some improvements to the consumption part of Kafka: without violating ACID transactionality, a partition (partition) is repartitioned within a consumer process based on a subpartition ID (hereinafter referred to as id) and TxType, and messages from the same subpartition are consumed serially, while messages from different subpartitions are consumed in parallel. The messages of the same sub-partition are consumed serially and the messages of different sub-partitions are consumed parallel. The figure shows Kafka parallel consumption.



Improvements to the Kafka parallel consumption model

- The message id is utilized as the value of the Key field of the Kafka message by default, and the product engineer is allowed to customize the id of the message, but the value's differentiation effect cannot be less than that of the Topic + Partition of the message.
- After the consumer process receives the message, the distributed transaction framework will first parse the metadata of the message to get the TxType and id of the message.
- The message is then repartitioned by “TxType + id” and automatically allocated and sent by the framework to a memory queue and processing unit for actual consumption by the business code.
- Messages with different “TxType + id” are assigned to different memory queues/processing units, which do not block each other and execute in parallel (or concurrently), and the degree of parallelism (concurrency) can be adjusted on the fly.

- As partitions are subdivided, the consumption progress defined on consumption groups and partitions requires an additional aggregation step to ensure that messages before a given offset are processed by the time Kafka sends an ack.
- The maximum length and maximum parallelism of memory queues/processing units can be configured and resources are reclaimed after a period of idleness to avoid memory buildup.

5.3.2 Deployment Details

- Released as a code base: Instead of introducing standalone services, Saga and Kafka-related logic is extracted into a public code base that is released on a version-by-release basis, deployed and upgraded along with the services located in the combinatorial orchestrator layer.
- Producers and consumers coexist in the same process 1:1: For services that need to initiate and manage distributed transactions, each node starts a producer and a consumer, and with the help of an existing cluster deployment tool (Amazon EKS), all nodes of the service are guaranteed to be connected to each other and to Kafka. This deployment allows us to call back the producer node directly from the consumer node without introducing additional message buses or other data-sharing mechanisms. Later, producers and consumers can be deployed on different services as needed, as long as their nodes are interconnected.
- Both Kafka and Go channel queueing modes are supported: When using Kafka queueing mode, the system conforms to the definition of ACID, while using Go channel queueing mode only guarantees A in ACID, not I and D. Go channel mode can be used during development and unit testing and is generally used during service integration testing and online deployment. Kafka mode is generally used for service integration testing and online deployment. If the online Kafka service is not available as a whole, the service that initiates distributed transactions can be downgraded to Go channel mode.
- Shared Kafka topic and partition: Multiple services or processes can share Kafka topic and partition, use consumer groups to differentiate consumption progress, and use TxType to do event triage.

5.3.3 System Availability Analysis

The high availability of a distributed system relies on each service involved being robust enough. The following is a categorized exploration of the various services in a distributed transaction, describing the availability of the system when some of the service nodes fail.

- Producer failure: The producer is deployed with an organization/orchestrator service with node redundancy. If some nodes of the producer's service fail, clients will see requests fail or time out for all transactions on that node that send queue messages and have not yet received callbacks, which can be successfully committed after retrying diversion to a normal node.
- Consumer failure: Consumers, like producers, are deployed with the organization/orchestrator service with node redundancy. If a consumer's part of the node fails, the client will see the request timeout for all transactions on that node that have received a queue message and have not yet sent a callback and Kafka will mark the consumer offline after the configured consumer session timeout (default is 10 s, can be customized per consumer), and then load adjust the topic and partition to distribute as evenly as possible by some algorithm to the remaining consumers in the current consumer group. The load is then adjusted to the topic and partition and distributed as evenly as possible to the remaining online members of the current consumer group according to a certain algorithm, and the load adjustment time is typically in the order of seconds. From the time the consumer's node fails until the end of Kafka load tuning, the messages on the topic and partition that the failed consumer is responsible for cannot be processed during this time. Customers will see timeout errors for some requests. Retries with the same data will also fail during this time if the submitted data has a direct mapping to the partition that generated the queued message.
- Domain service failure: A given distributed transaction will rely on multiple domain services, each deployed independently with redundant nodes. If some nodes of a basic service fail, the corresponding request of the distributed transaction will partially fail at the corresponding step, and the preceding steps will be executed in sequence to compensate for the interface. The customer sees a timeout or a failure message customized by the business, and a retry is likely to succeed. Businesses can introduce a service fusion mechanism to avoid message buildup.
- Message queue failure: Kafka itself has master-slave replication, node redundancy, and data partitioning to achieve high availability, which is not discussed in depth here.

5.3.4 Production Issues and Handling

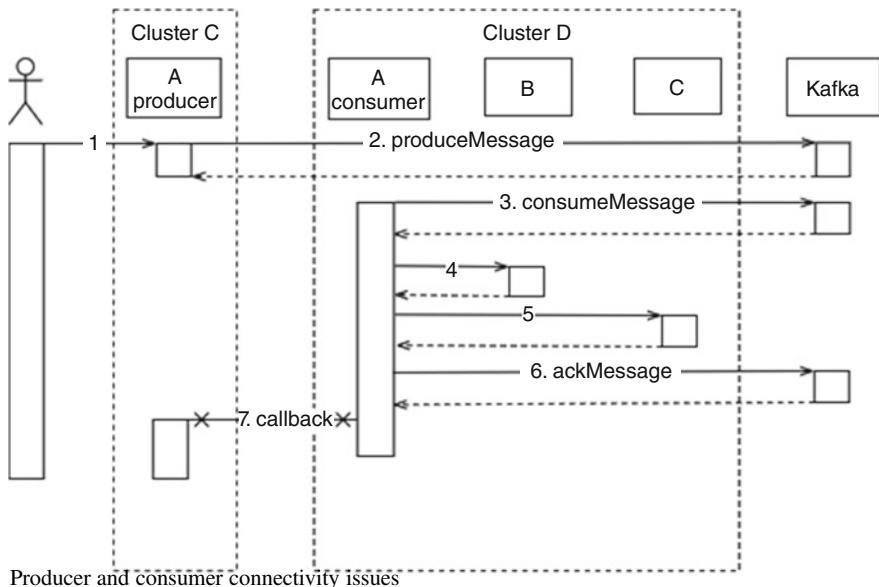
After the distributed transaction framework was released with the service, it ran online for a period of time and basically met the design expectations. There are some issues that have arisen during this period, which are listed below.

1. Producer and consumer connectivity issues

A service using distributed transactions had a timeout on some data while other data returned normally, and client retries did not solve the issue. Analysis of the logs revealed that the messages were sent by the producer and processed by

the consumer successfully, but the callback from the consumer to the producer failed. Further study of the logs revealed that the node where the consumer was located and the node where the producer was located were in different clusters that are of separate networks. Looking at the configuration, the same Kafka brokers, topics, and consumer groups are configured for the same service in both clusters, and consumers in both clusters are connected to the same Kafka and are randomly assigned to process multiple partitions under the same topic.

As shown in the figure, service A (producer) in cluster C and service A (consumer) in cluster D use the same Kafka configuration. Although their nodes are connected to Kafka, they are not directly connected to each other, so the callback in step 7 fails. The reason why some data timeouts and retries are invalid and others are fine is that the value of a particular data is mapped to a particular partition, so if the message producer and the consumer of the partition are not in the same cluster, the callback will fail; conversely, if they are in the same cluster, there is no problem. The solution is to modify the configuration so that services in different clusters use different Kafka.

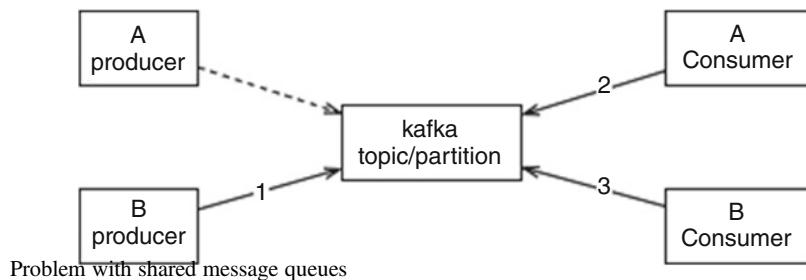


2. Problems with shared message queues

Service A has a business exception alarm about the consumer of a distributed transaction receiving a queue message of a type that does not meet expectations. By analyzing the logs and viewing the code, it is found that the message type belongs to service B and the same message has already been processed by a consumer of service B. Checking the configuration reveals that the distributed transactions of services A and B use the same Kafka topic and are configured with

different consumer groups to distinguish the progress of their respective consumption.

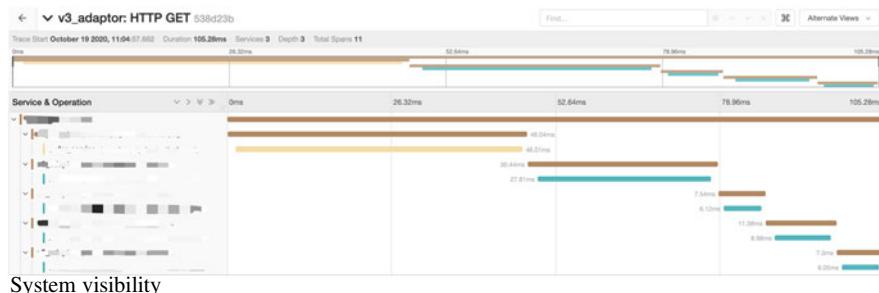
As shown in the figure, services A and B share the Kafka topic and partition, the exception message comes from service B's producer (step 1), the exception alarm appears at A's consumer (step 2), and B's consumer also receives and processes the message (step 3), with steps 2 and 3 running in parallel. The producer of service A has no role in this exception. There are two ideas to solve this problem: either modify the configuration to remove Kafka topic sharing or modify the logging to ignore unrecognized distributed transaction message types. Since the production capacity of service A+B on this topic is less than the consumption capacity in the short term, removing the sharing would further waste Kafka resources, so the modified logging approach is used for now.



3. System visibility improvements

One of the challenges of distributed systems is the difficulty in troubleshooting and isolating problems because the RPC call graphs are complex. The introduction of asynchronous queues for distributed transactions, where producers and consumers may be located on different nodes, requires a better solution about service visibility, especially request tracing.

For this reason, the distributed transaction system of FreeWheel's core business system adopts the tracing system, which visualizes the flow of distributed transaction data across services and it helps engineers to pinpoint functional and performance problems, as shown in the figure.



In addition, We can use Kafka's multicast feature to browse and replay messages at any time using temporary consumer groups, which does not affect the normal consumption of data.

4. Loss of business exception details

A service using distributed transactions found that the customer had a steady 5xx error when committing specific data, and retries were ineffective.

After analyzing the logs, it was found that a basic service returned a 4xx error for that data (the business thought the input data was not valid), but after the exception capture and processing of the distributed transaction framework, the original details were lost and the exception was rewritten to a 5xx error before being sent to the customer.

The solution is to modify the exception handling mechanism of the framework to aggregate the raw exception information encountered at each step in the consumer process, package it into callback data, and send it to the producer, allowing the business code to do further exception handling.

5. The domain service creates duplicate data

Service A, which uses distributed transactions, finds that occasionally a request is successful, but multiple entries of the same data are created in the database managed by domain service B.

It was discovered through FreeWheel's tracing system that service A called B's creation API and retried on timeouts, but both calls succeeded at service B and the interface was not idempotent (i.e., the effect of multiple calls is equal to the effect of one call), resulting in the same data being created multiple times.

Similar problems arise frequently in microservices practice, and there are two ways to solve them.

- One solution is final, i.e., A and B share the timeout configuration; A passes its own timeout setting t_A to B, and then B commits the data transactionally according to a timeout t_B that is shorter than t_A (taking into account the network overhead between A and B).
- Another way to solve the problem is to implement idempotency in service B's interface (this can be done by setting unique keys in the database, creating data requests that require unique keys, and ignoring requests with conflicting keys).

Regardless of whether distributed transactions are used, the problem of duplicate data being requested multiple times by the client due to network retries is a practical issue for every micro-service, and implementing interface idempotency is the preferred solution.

5.4 Chapter Summary

This chapter discusses the concepts, technical approach, and practices of distributed transactions based on our team's experience and practice. We introduced a distributed transaction solution that supports heterogeneous databases with eventual

consistency and discussed the issues encountered after this solution went into production and the attempts to solving these issues.

As mentioned in the background section of this chapter, distributed transactions with eventual consistency guarantee are necessary for microservice systems, in order to bridge the divergence between the state of the business models and the state of the storage models, adapting to changing business requirements and continuous adjustment of service boundaries. However, the procurement of distributed transactions comes at the cost of adding dependency on queues and the maintenance effort of interfaces for compensating data, as well as higher requirements for the idempotency of interfaces. If the discrepancy between logic and data could be eliminated at the root, i.e., by realigning the boundaries of the storage model with the business model, it would certainly be more straight-forward and resilient to encapsulate transactions to a single microservice and have the traditional ACID transactions supported by a relational database. Moreover, if for some business cases, the business models and storage models diverge but with less impact and frequency, we can also opt out to adopting distributed transactions, collect exceptions through logging and monitoring systems, and handle the exceptions in batched task queues. After all, there's no silver bullets for software development, and our systems (and ourselves too) must embrace change and move agile to stay relevant.

Chapter 6

Serverless Architecture



As the most influential organization in ICT, the Computing Technology Industry Association (CompTIA) announces the ten most influential emerging technologies every year. In the 2020 study, serverless architecture (hereafter called Serverless) was selected to the list for the second time, coming in fourth behind AI, 5 G, and IoT technologies.

Compared with AI, 5 G, and IoT, which have been recognized as hot concepts in recent years, Serverless is not mentioned so frequently because it is more of developers and cloud service providers-oriented concept, but the fact that it has been selected for 2 years shows that its momentum and contribution to information technology is becoming stronger and stronger.

So, as a new application architecture, what is Serverless; what are its features, advantages, and application scenarios; and what changes does it bring? This chapter will answer these questions.

6.1 What Is Serverless Architecture

When you hear this word, you may be a little confused. Even for cloud-based services, their underlying operations still need to be executed by physical servers, so how is it possible that applications can work without servers?

In this section, we provide insights into the definition and development of serverless architecture, as well as its advantages and disadvantages.

6.1.1 Definition of Serverless Architecture

Like many other concepts in software design such as microservices, Serverless does not have a recognized and clear definition.

Ken Fromm, the VP of Iron.io, and Martin Fowler, the father of software development, both have their definitions. Here, we quote the definition from a white paper on Serverless published by CNCF in 2018.

Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.

There are two important roles involved in Serverless:

- Developer: the person who writes the application, as a user of the service, concerned with the design and implementation of the application
- Cloud service provider: the provider of the application runtime environment, responsible for providing computing and storage resources

Applications cannot run without servers (whether physical, virtual services, or containers), and Serverless is the same. However, what cloud providers can do is abstract server-dependent resources such as runtime environment, computing and storage, security, and auto-scaling requirement into various services to achieve complete transparency to developers, and thus the latter would have a “serverless” experience.

Therefore, a more accurate understanding of the “serverless” concept in Serverless is that it is a service provided by cloud providers that allow developers to not care about servers.

As defined in the CNCF white paper, Serverless can be further divided into two categories.

- Backend-as-a-Service (BaaS): Cloud service providers use unified APIs and SDK to connect developers’ mobile or web applications to backend cloud services and provide public services such as account management, file management, or message pushing.
- Functions-as-a-Service (FaaS): The developer takes a function as an entry point, packages and uploads the code, and specifies the events that trigger the function, such as an HTTP request. When no event occurs, the function is not executed and no cost is incurred. When the specified event occurs, these functions are triggered, executed, extended, and destroyed automatically when the event ends. The developer pays only for the compute time used.

BaaS and FaaS both shield the underlying servers from developers, with the difference that the former only provides some standardized reusable software services, while the latter provides real computing and storage resources that developers can use to run their code.

FaaS is more like a revolution to the existing application architecture and has attracted more attention and discussions; we will focus on that part.

6.1.2 Development of Serverless Architecture

The concept of Serverless has been around for a long time, and it didn't emerge by accident. It evolved from the back-end architecture over a long period.

Before the concept of cloud computing was created, enterprises that want to build back-end applications not only need physical machines but also need to build network devices and data centers, set up security configurations, etc. Although the emergence of the Internet Data Center (IDC) largely reduced the burden on enterprises, the maintenance of these facilities requires professional staff, which still brings a considerable cost. In the process of building a server from scratch, it is almost impossible to avoid out-of-order construction and wasting resources. Even with a perfect and systematic guidance, it is difficult to make rapid responses and flexible adjustments in the face of the rapid increase in economic development and the complex changing needs.

The emergence of cloud service has greatly changed this situation. Cloud service providers consolidate and manage clusters of physical machines through dedicated hardware and software technologies, use virtualization technology to slice and package computing and storage resources on a granular basis to ensure security and reliability, and then provide them to developers on demand. For enterprises, they can set up a complete set of secure and reliable server facilities at a much lower cost and complexity, and all of this is on-demand and can be flexibly adjusted as demand changes.

Cloud services have different forms at different stages of development. The earliest IaaS (Infrastructure as a Service) is the most basic cloud service, which only provides some server resources that developers can rent according to their actual needs, but the installation, configuration, upgrade, and monitoring of the operating system in the server still need to be managed by developers.

To free developers from the management of servers, PaaS (Platform as a Service) was born. It runs on top of IaaS and takes over the management of software such as operating systems and provides an automated deployment platform so that developers only need to focus on application development.

By this stage, developers have broken out of server management but, still, need to configure servers in advance. Even with the introduction of Kubernetes for container management, it is still necessary to assign a specified configuration and number of servers as nodes for the cluster to achieve elastic scaling limited by the cluster size.

For a long time, developers want to scale and be charged based on traffic without additional cost, and so Serverless was born.

The concept of Serverless can be traced back to the article “Why the Future of Software and Apps Is Serverless” published by Iron.io’s VP, Ken, in 2012. In this article, Ken expressed his views on Serverless.

Even with the rise of cloud computing, the world still revolves around servers. That won’t last, though. Cloud apps are moving into a serverless world, and that will bring big implications for the creation and distribution of software and applications.

Ken was a big fan of Serverless at the time, but this concept came to most people's attention 2 years later.

In 2014, the world's first FaaS service, AWS Lambda, came out of nowhere. In the following years since then, major cloud providers launched their own FaaS services, including IBM OpenWhisk on Bluemix, Google Cloud Functions, and Microsoft Azure Functions.

Domestic cloud service providers such as Ali Cloud, Tencent Cloud, and Huawei Cloud entered late; however, with strong technical strength, they have quickly jumped into the first echelon. Ali Cloud even stood out in the FaaS platform assessment report released by Forrester in 2021, becoming the industry leader second only to Amazon and Microsoft.

It's safe to say that as the demand for Serverless grows and cloud providers become more supportive, Serverless is playing an increasingly important role, and it's only the beginning.

6.1.3 Advantages of Serverless Architecture

There are many benefits to choose Serverless as your application architecture, including lower costs and faster product iterations. In summary, Serverless offers the following advantages.

1. Lower Cost

There are two aspects.

The first aspect is charging according to the actual usage. Whether it is the IDC or cloud-based IaaS, or PaaS services, the billing method is essentially a monthly payment, and the specific amount of the fee is only related to the server configuration and the length of the lease. In other words, even if the application is not accessed by users most of the time, we still need to pay for it monthly. To make it worse, to meet the demand of handling peak loads, we usually choose servers based on maximum resource usage, which drives up the cost of renting servers even more. This problem does not exist when choosing Serverless. For a FaaS service, the application is initialized and executed only when a request arrives and is destroyed when the request returns. Also, we only pay for the amount of time that the request is processed. Especially when there is no request, the application is not running, and, of course, there is no need to pay. This charging model is perfect for applications that do not have high server utilization but need to be able to handle bursts of traffic in real-time.

The second aspect is zero server maintenance cost. For most companies, server management and maintenance costs are a big expense, and the cost of hardware and software and professional maintenance personnel often takes up a significant part of the expenditure. With Serverless, the deployment can be achieved at zero cost of server operation and maintenance.

2. Automatic Scaling and Expansion

To handle the possible sudden increase in traffic, traditional server architecture will always reserve some server resources to act as a buffer. These operations more or less require human intervention, while the granularity of scaling is often limited by the level of individual services, and the number and speed of scalability are limited by the size and configuration of the cluster. With Serverless, each API is a FaaS service, and its operating environment and configuration can be specified separately, and the granularity of scaling is accurate to the API level; the parallel processing power provided by FaaS is also adjusted according to the traffic, which is a native feature of FaaS, automatically implemented without effort from developers. With proprietary hardware and software, the scaling speed can be as short as milliseconds, and the scaling limit is no longer limited by the cluster size.

3. Faster Iteration and Delivery

For some enterprises that are not sensitive to server costs, the biggest advantage of Serverless still lies in its ability to significantly reduce iteration cycles and enable faster delivery.

Serverless shields developers from a lot of the complex details of deployment and operation, and what used to take weeks for several engineers to complete can be done in minutes with Serverless. Also, to attract more developers, Serverless providers often offer a rich set of services to support application scenarios. These services may include database services, indexing services, logging services, validation services, file services, etc. The advantages of FaaS services are fully embodied, as they can be easily, efficiently, and securely integrated with these services. Users don't even need specialized background knowledge, just a simple drag and drop on the UI to bring online a secure and reliable web service containing database access, authentication, and other features in a matter of minutes.

On top of that, powerful cloud providers have professional teams behind them that specialize in performance optimization, security construction of the operating environment, and other basic services that make Serverless services less risky. API granularity of service slicing makes it more flexible and easier to integrate with other systems. If you want, you can even keep the original system running and only use Serverless to perform certain specific tasks.

6.1.4 Shortcomings of Serverless Architecture

Migrating the whole or a part of a system to Serverless can bring many benefits, but that doesn't mean that Serverless can handle all scenarios well. Before the migration, you need to carefully consider these aspects as follows.

1. End-to-End Testing Difficulties

For developers, the first problem they may face is the difficulty of end-to-end testing. Traditional services perform their tasks as processes, and it is easy for

developers to build such a runtime environment locally and observe and debug its behavior. Even when compatibility issues are encountered with local system, they can be dealt with by containerization or virtualization technology. However, for Serverless services, the application is just a piece of code without context and usually cannot run on its own. This piece of code works only when it is loaded by the cloud service platform. The details of this process are often not published for security reasons.

Some cloud providers offer containerization solutions for their services, but it is not easy to run their companion services locally. It usually ends up with integrating in a separate set of environments provided by the service provider and only running unit tests locally. This introduces a new problem: for cost reasons, one developer often cannot have a whole set of exclusive environments; any action may break each other, which undoubtedly brings non-negligible problems to parallel development and cooperation in teams.

2. Cold Start

Serverless offers a pay-per-use model because the service is only initialized and runs when a request arrives. That is, from the time the first request arrives until that request is processed, there is a series of processes, such as node selection, code download, and initialization of the runtime environment. Therefore, the first request will take longer to process, and subsequent requests will not encounter this problem.

3. Not Suitable for Long-Time Operation

One important reason for the low cost of Serverless is that costs only happen when requests arrive, which means the longer the server is idle, the more savings Serverless can achieve. For applications that run jobs or handle requests for a long time, Serverless may not be able to reduce the cost, and IaaS or PaaS may be a more economical option. Besides, the most Serverless will limit the executable file size and execution time of each function, such as AWS Lambda does not allow a lambda to run more than 15 min per request.

4. Vendor Lock-In

When you start thinking about using Serverless, the problem you have to face is how to choose a good cloud provider, and especially when you already have a large infrastructure and your applications have reached a large scale, it will be a difficult choice. Migrating your system to Serverless also means that you have to hand over the stability, scalability, and security of your service to the cloud provider. Once you use a cloud provider's Serverless service, in most scenarios, you also need to use other services provided by that cloud provider, such as file storage, database, message queue, etc., that is, the application is locked to the cloud provider. Once you change the cloud provider, this will become a huge project.

5. Restricted Operating Environment

Serverless hides the details of program execution through multiple layers of virtualization and abstraction to provide consistent services to different tenants. Correspondingly, the tenant loses absolute control over the configuration. The tenant can only choose from the features and configurable parameters provided by

the cloud provider and may be limited in the version of the programming language, the size of the executable file, and disk space quota. At the same time, direct access to physical hardware or modifications to the operating system configuration becomes difficult, tuning tools based on hardware and software coordination become unavailable, and troubleshooting becomes very difficult.

In addition to these generic limitations, each cloud provider has its specific limitation. Developers considering using Serverless can only make the right decision that best suits the application scenario once they have a clear understanding of its strengths and weaknesses.

6.2 Serverless Architecture Applications

Theoretically, all traditional server application scenarios can be implemented using Serverless. However, Serverless itself is not a one-size-fits-all solution due to its various characteristics.

Serverless is just a new cloud service that allows developers to focus on business logic and enable rapid deployment, and it is better suited to solve specific problems such as tasks triggered by specific events or timings, low frequency of requests, or scenarios with bursty traffic.

According to the characteristics of Serverless, we summarize several typical scenarios of applying Serverless.

6.2.1 *Building Web API Backend Services*

The most common use scenario for Serverless is as the backend of a web or mobile application, providing highly available and scalable API services to the front end.

In terms of quality of service, Web APIs are attracting higher user expectations and requirements. These requirements include not only the ability to provide a consistent, secure, and reliable user experience, provide timely feedback, and make rapid adjustments but also the ability to provide high availability and to handle bursty traffic.

In terms of implementation, a single call to the Web API often requires multiple services to work together, such as DNS, gateways, caching services, databases, message queues, etc.

These features make a seemingly simple Web API service usually cost much more human resources. Serverless makes this process extremely simple and efficient. The architecture of a Serverless-based Web API backend is shown in Fig. 6.1.

Three are three main processes.

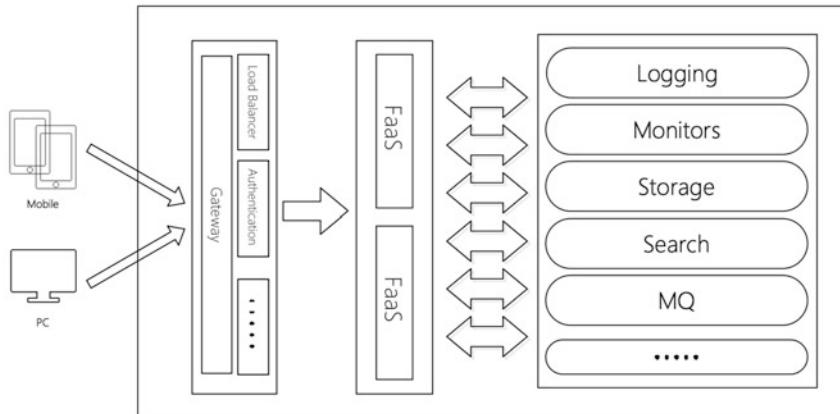


Fig. 6.1 Serverless based web API Backend Architecture

- The web or mobile page sends requests to the service address, usually via standard HTTP.
- Requests are resolved to a certain gateway address, which serves as a unified entry point for backend applications. The gateway is responsible for implementing tasks such as traffic management, CORS support, authorization, access control, and monitoring and, the most important, triggering different FaaS services based on different request paths.
- FaaS runs business code and performs calculations based on incoming requests. Different services are called according to different user scenarios to implement file reading and writing, data persistence, message distribution, etc.

Throughout the process, developers only need to focus on FaaS code implementation, and the rest of the services can be provided by cloud providers. These services are often available out of the box, and developers can even build and integrate the services with just a click and drag on the UI.

Another important feature of the Serverless Web API is the flexible and small granularity of service segmentation, which makes it ideal for replacing legacy APIs or acting as a proxy for legacy APIs.

In practice, we often encounter situations where we migrate old systems to new systems and technology stacks to meet increasing performance and functional requirements. This process may encounter a scenario where most of the features can be easily migrated, but there are always a few legacy features that cannot be migrated well due to some minor incompatibilities between the old and new technologies. For these legacy features, we have to continue to maintain a large legacy system. Serverless provides the flexibility that new systems or technology stacks do not have, supporting fine-grained service splitting, which makes migration at API granularity possible. Serverless can also be used as a proxy for legacy APIs, converting customer requests and responses into the format supported by legacy functionality. Each of these usage scenarios can effectively reduce the maintenance costs of new and legacy systems.

Building Web API services based on Serverless is inherently highly available and scalable, and other services that accompany it, such as databases, message queues, file services, etc., are often implemented based on a distributed architecture, which saves a lot of time and labor costs while ensuring high availability. Mainstream cloud service providers also allow developers to deploy their services to service nodes in different regions of the world so that users in different geographical locations can get a good user experience. In addition, cloud service providers offer a wealth of complementary services to address various scenarios, and FaaS services can be efficiently and securely integrated with these services.

These features make it increasingly attractive to build Web API services based on Serverless.

6.2.2 Building the Data Orchestrator

In addition to running business code directly, Serverless has a very common use scenario—data orchestration.

Whether you are focusing on front-end or back-end development, you are certainly familiar with the MVC design pattern. In the classic MVC pattern, M refers to the business model, V refers to the user interface, and C refers to the controller. The purpose of using MVC is to separate the implementation code of M and V so that the same program can have different presentation forms.

As user interaction becomes more and more complex, V gradually developed into a single-page application, and M and C gradually developed into service-oriented programming SOA back-end applications. After the separation of the front and back ends, the BFF (Backend for Frontend) layer is usually introduced to do data orchestration to achieve multi-end applications and multi-end business decoupling.

Because the BFF layer is only used to implement stateless data orchestration, which fits well with the design philosophy of FaaS, an architecture as shown in Fig. 6.2 is formed, called SFF (Serverless for Frontend).

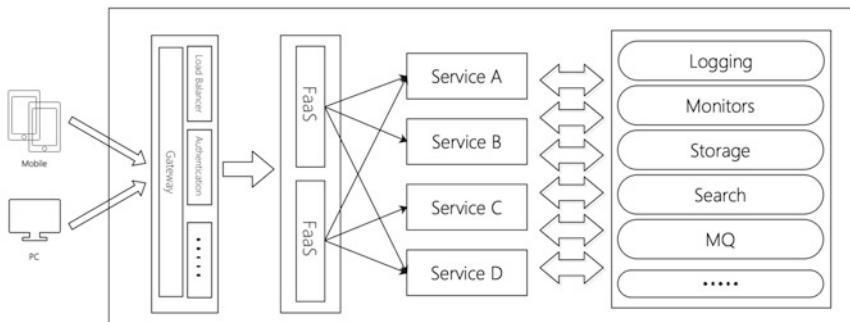


Fig. 6.2 Serverless for frontend (SFF) architecture

Under this architecture, the request from the front end reaches the gateway and triggers the FaaS service, which invokes the metadata endpoints provided by the back end to reorganize the data and form the data structure needed by the front end.

More importantly, the effort to build a highly available FaaS service from scratch is quite low, and the service users and data consumers are fully capable of building a data orchestrator that meets the requirements in a short time. In this way, the work of customizing the Web API of front-end and back-end services is transferred from service developers to service users; the former can focus on the implementation of the business domain model, while the latter can implement data orchestration according to their needs flexibly, which greatly enables faster delivery and iteration of front end and back end.

6.2.3 Building Timed Tasks

The FaaS services provided by Serverless can be triggered in two ways: event triggered and timed triggered.

The event-triggered approach is the most common one. As in the previous two sections, when FaaS is used as a Web API service, a user request can be considered an event. When using FaaS to listen to a database source, a modification to a record is also an event. The types of events supported by FaaS may vary depending on the cloud provider.

The timed-triggered approach is more suitable for scenarios where certain repetitive tasks are performed on a scheduled basis. Different cloud providers offer different triggering methods, such as that AWS can use the standalone companion service CloudWatch Events to create triggering rules, while AliCloud has triggers built into the FaaS service. Whichever way you choose, users can easily make their own rules.

When a rule is created and triggered at a certain point, its corresponding FaaS service is activated and starts running the specified task. Such tasks that need to be triggered at regular intervals include but are not limited to the following categories.

- Web crawlers that regularly crawl the web for data
- Regular backup or import and export of data
- Regular deployment in automated operations and maintenance
- Regular data collection and analysis

Compared with traditional architecture, Serverless emphasizes modularity and high availability by organizing and arranging different functional high availability modules to form a complete system, while the modules can be flexibly replaced, added, and removed.

Next, we take Fig. 6.3 as an example of how to build a web crawler application based on Serverless.

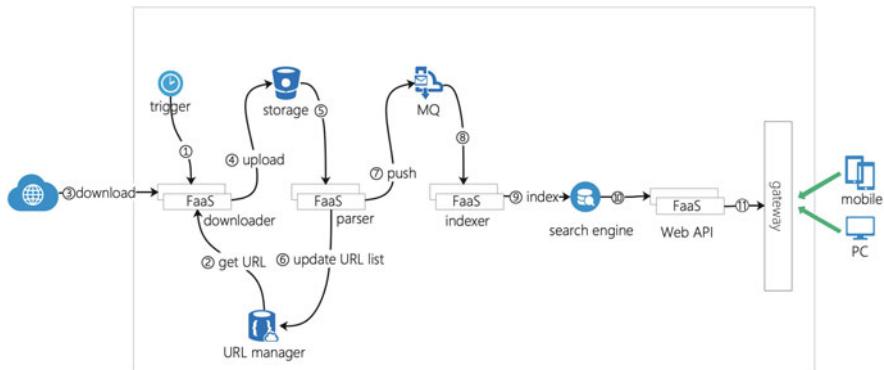


Fig. 6.3 Building a web crawler application with serverless

- First, you need to create a timed trigger that activates the FaaS-based downloader at a fixed time of day (e.g., midnight); see step (1).
- The downloader reads the seed URL from the URL manager, downloads the web data corresponding to the URL, and uploads it to the file storage service; see steps (2)–(4).
- When the web data is uploaded to the file storage service address, it triggers the execution of the downstream web parser built by the FaaS; see step (5).
- The parser downloads the data of the page and extracts the needed data and the new URL; the data is pushed to the message queue, and the URL is added to the list of existing URLs; see steps (6) and (7).
- After the data is pushed to the message queue, the downstream FaaS service-based indexer is triggered, which arranges the data into a structure that can be queried and stores it in a database or search engine; see steps (8) and (9).
- Use FaaS to build a Web API backend and integrate gateway services to expose CRUD and other interfaces to the outside world; see steps (10) and (11).

Of course, a complete crawler system involves many more technologies, such as URL filtering and de-duplication, website authentication and login, dealing with anti-crawler measures on websites, efficient parallel downloading of web pages, dynamic web crawling, and so on. This architecture only shows the basic modules involved in a crawler system and how to build them using Serverless. In the real implementation, it needs to be adjusted according to the actual requirements, such as adding caching services, monitoring systems, etc.

6.2.4 Building Real-Time Stream Processing Services

Streaming data is data that is continuously generated by thousands of data sources and sent in the form of records. The size of a single record is usually small, but, as it

accumulates over time, the total amount of data can often reach the TB level or even PB level. Real-time stream processing refers to the processing of streaming data in real time and continuously. Compared to offline computing, real-time stream processing provides faster data presentation and shorter time delays.

Through the development of big data technology in recent years, real-time stream processing is gradually becoming a major trend in the future, such as the emergence of more and more online applications such as live streaming and short video, and online machine learning and real-time data warehousing are becoming more and more popular.

Application scenarios for real-time stream processing include, but are not limited to, the following:

- Financial institutions collect users' transaction data in real time for user behavior analysis, real-time risk control and anti-fraud detection, and alert and notify in the first place.
- Road and vehicle sensors in intelligent transportation continuously hand over data to data center, which collect, analyze, and aggregate these data to monitor traffic conditions in real time, help optimize and plan vehicle trips, and predict and inform road traffic risks in advance.
- The monitoring system continuously collects application information, draws system status dashboard, identifies abnormal status, and alerts in time.
- Video websites transcode and store videos uploaded by users in real time and automatically review them based on machine learning algorithms.
- The website collects users' browsing records in real-time, generates user trajectories and profiles, and optimizes content delivery algorithms to provide a better experience.

Currently, the popular stream processing systems are built on Apache Spark or Apache Flink, which is especially known for its high throughput and low latency. It is not difficult to build such a system according to the official documentation, but, if you want to use it in a production environment, how to ensure high reliability, scalability, and maintainability will turn out to be a big challenge.

Compared with traditional solutions, the real-time stream processing system based on Serverless is naturally highly available, scalable, and easy to build. Its automatic scaling can cope with sudden increase in traffic and effectively reduce the maintenance cost of the system during idle time. Figure 6.4 shows a complete process of implementing a real-time stream processing system based on Serverless.

The above process can be divided into three stages, as follows:

Phase I: Data Collection

- Streaming data is usually collected from thousands of different data sources, which can either be sent directly to the streaming storage service via the SDK or converted to a simple format by FaaS service before being sent out; see step (1).
- The stream storage service is responsible for receiving and storing this data and acting as a buffer for downstream consumers.

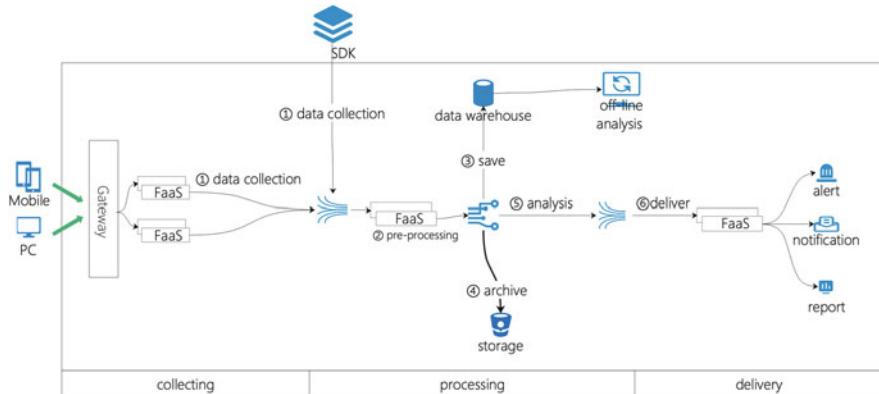


Fig. 6.4 Real time stream processing system with serverless

Phase 2: Data Processing

- Once the stream storage service captures the data, it pushes this data to the downstream stream processing service. In order to provide a unified format to the stream processing service, this data is usually processed by the FaaS pre-processor in advanced, such as checking data consistency, handling invalid data and missing values, etc.; see step (2).
- After receiving these data, the stream processing service does not consume them immediately but persists them first for later query and extracts the interested data and then pushes them to the data warehouse for later offline computation; see steps (3) and (4).
- The stream processing service performs data analysis and aggregation on demand, such as performing time-range-based metric calculations, generating aggregation results for real-time monitoring dashboard, and monitoring exceptions through machine learning algorithms and historical data; see step (5).
- The processed data will be sorted and pushed to the downstream stream storage service.

Phase 3: Data Delivery

- FaaS subscribes to different topics in the streaming storage service and delivers data to different consumers according to different topics, e.g., pushing abnormal data to the alarm system; see step (6).

As can be seen in Fig. 6.4, the point of the Serverless-based real-time stream processing system is to use stream storage service and stream processing service as the center and FaaS as the glue. Developers can not only use FaaS to implement business logic but also use it as a bridge to connect various services, which makes the system extremely flexible and scalable, greatly simplifies the programming model, and improves the building efficiency.

6.3 Serverless Architecture in Practice

Through the introduction of the first two sections, I think you have a clear understanding of the concept of Serverless and its application scenarios; in the following section, we will introduce a few cases based on AWS Lambda services and hope it can bring you some inspiration.

6.3.1 Why AWS Lambda

There are two main ways to build Serverless.

The first is to build your own Serverless platform with the help of open source projects such as Knative and OpenFaaS. However, this approach is costly usually and requires a lot of developers and operators.

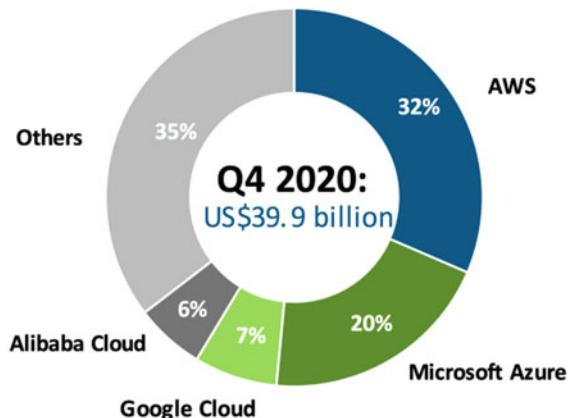
The second is to use out-of-the-box FaaS services provided by cloud providers directly, which can save a lot of build and maintenance costs. This approach is more friendly for teams that have tight project timelines and lack the expertise to build a Serverless platform, which is the reason why we chose it.

For the second approach, the first thing to do is to choose a suitable cloud provider.

According to the report “Global cloud services market infrastructure market Q4 2020” published by Canalys, AWS, Microsoft Azure, Google Cloud, and Alibaba Cloud hold 65% of the global market share, as shown in Fig. 6.5.

AWS Lambda, the world’s first Serverless service, was a game changer for cloud services and has been seen as synonymous with Serverless from its launch. We chose AWS Lambda for several reasons.

Fig. 6.5 Global cloud services market share



- High market share: AWS Lambda has the largest and most vibrant community, with millions of active customers and thousands of partners worldwide, a broad user base, and a rich set of use cases.
- Support more programming languages: Common programming languages, including Java, Node.js, and other providers' products, such as Microsoft Azure Functions, support five programming languages. AWS Lambda also allows users to customize the runtime environment, which can theoretically support all programming languages.
- Rich and easily integrated packages: Offering more than 200 services, from infrastructure technologies, such as compute, storage, and database, to emerging technologies such as machine learning, artificial intelligence, data lakes, and the Internet of Things, AWS offers more services and more features within them than any other cloud provider, addressing a wide range of usage scenarios and providing many different solutions.
- Rich and high-quality documentation: Almost all the problems encountered during the development process, including concepts and principles, usage, and examples, can be found in the documentation.
- More data centers all over the world: 80 available service centers operating in 25 geographic regions around the world, fast and stable service is provided worldwide.

As facing the problem of cloud service provider lock-in, once you choose a certain cloud service provider, it is almost impossible to change it in the late. Therefore, you must do a thorough research to clarify your needs and the characteristics of different cloud service providers. A mature and experienced cloud service provider can make the road to the cloud smoother. When the cost gap is not particularly large, priority should be given to a mature and stable cloud provider. After all, the cost difference often seems insignificant compared to an online incident.

6.3.2 Import and Processing of Large Amounts of Data

In practical development, we may encounter scenario where there is a large amount of data to be imported, and this data, due to its large volume, is often provided in the form of files. Such requirements are usually difficult to handle due to the file operations involved. In this section, we will introduce a system for importing and processing large amounts of data built on AWS Lambda. The user requirements are generally as follows.

- Data import is performed once a week.
- The amount of data imported is about three million items.
- The data is provided as 4 CSV files, 3 GB in total.
- The imported data needs to support updates, multidimensional retrieval, and export functions.

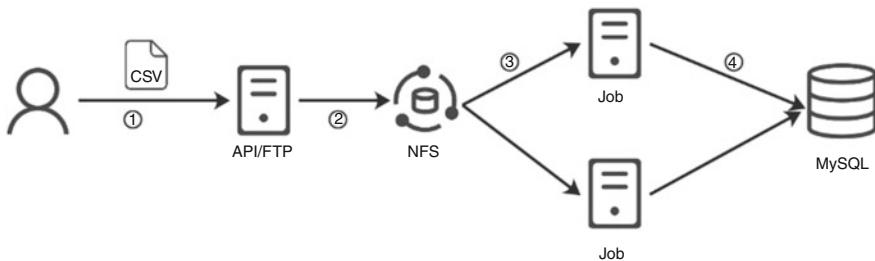


Fig. 6.6 Traditional solution architecture of processing data

To address these needs, we first consider the traditional solution, as shown in Fig. 6.6.

The user accesses the server through API or FTP service to store the file to the Network File System (hereinafter referred to as NFS), as shown in steps (1) and (2). After the file is uploaded successfully, the execution of the downstream background task is triggered by message queue or file listener, as shown in step (3). The background task is responsible for downloading, parsing, and storing the structured data into MySQL database as shown in step (4). MySQL is responsible for persisting the data and providing retrieval and export functions.

A few key points here are as follows:

- Use NFS as the file storage system.
- Background tasks are triggered by event or by schedule.
- Use MySQL as the database system.

The advantage of this solution is that it is more common and the related technologies involved are more mature. However, the disadvantages are also obvious. Next, we discuss the problems and how to design the system based on Serverless.

1. File Storage System

NFS is bound to the physical, which is transparent to the application but reduces the scalability and maintainability, and requires specialized personnel to build and maintain. Also, reading and writing to NFS also requires service processes to cooperate. That is, we need to provide a separate API interface or build FTP services to achieve file transfer from the user side to the NFS side.

We just simply need a service to store user files temporarily, but it creates so much extra burden, which can be solved by using Amazon S3 (hereafter S3). S3 is a fast, reliable, and scalable storage service that supports to transfer and share data based on a RESTful API or SDK toolkit and also supports cross-account file sharing. Customers can easily import files through cross-account file sharing using AWS's managed services.

In order to implement it, we only need to perform a very simple configuration. In the following example, we will grant access to users in the customer's AWS account (Account B), thus allowing these users to upload data to the system account (Account A), as shown in Fig. 6.7.

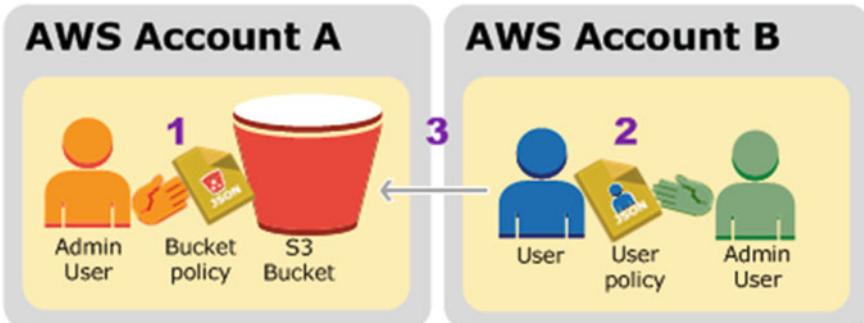


Fig. 6.7 Granting access to AWS accounts (Account B and Account A)

First, we need to create a bucket in account A. A bucket is a container that S3 uses to store files. Each file needs to be stored in a bucket, which is similar to a folder in an operating system. We need to attach a policy to this storage bucket, which is responsible for specifying the following:

- The name of the shared bucket, such as bucket-to-be-shared
- The ID number of account B, e.g., 1122334455, and the name of a user under that account, e.g., sandman
- The list of permissions required by the user

The policy is configured as follows:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "allowotheruser",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::1122334455:user/sandman"
      },
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3>List*"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-to-be-shared",
        "arn:aws:s3:::bucket-to-be-shared/*"
      ]
    }
  ]
}
```

Then, account B grants the bucket-to-be-shared access policy to user sandman, configured as follows:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Example",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3>List*"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-to-be-shared",
        "arn:aws:s3:::bucket-to-be-shared/*"
      ]
    }
  ]
}
```

This allows user sandman under account B to upload files manually or use access key to automatically export and upload files to account A's bucket.

Using S3 and its cross-account bucket sharing feature, the process can be completed in a short time with only UI configuration, saving you the trouble of setting up an NFS file storage system and simplifying the process of setting up FTP or a dedicated API for file upload.

2. The Way to Trigger Background Tasks

In order to trigger the execution of downstream tasks after file upload, upstream tasks usually notify downstream tasks via messages, or downstream tasks watch changes in file directories. Both of these methods need to reserve resources for downstream tasks, which will waste server resources when the load rate is low and make it difficult to auto-scaling when the load rate is high. If you choose the messaging approach, you also need to introduce an additional message queue component. Also, file processing and stored procedures take up high I/O resources, which may impact other services.

Once we choose S3, the Lambda would be our best choice for next. As we mentioned earlier, AWS Lambda can be run as an event-triggered application, and the file upload event will trigger and execute a Lambda instance.

First, we need to create a Lambda function `data_ingestion_handler` and choose an appropriate programming language; here, Golang is chosen, as shown in Fig. 6.8.

After Lambda is successfully created, you also need to add event trigger rules for this Lambda and use the file creation operation in S3 storage bucket as the trigger event, as shown in Figs. 6.9 and 6.10.

Once this step is done, S3 triggers the execution of the Lambda function `data_ingestion_handler` once a file is uploaded to the storage bucket `bucket-to-be-shared` and passes the following structure as an argument to the Lambda function.

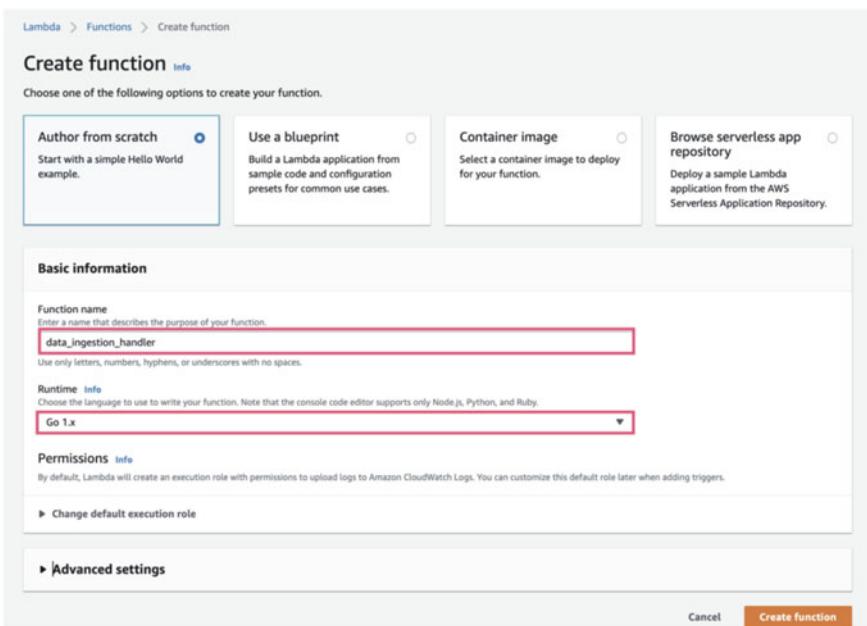


Fig. 6.8 Creating lambda function

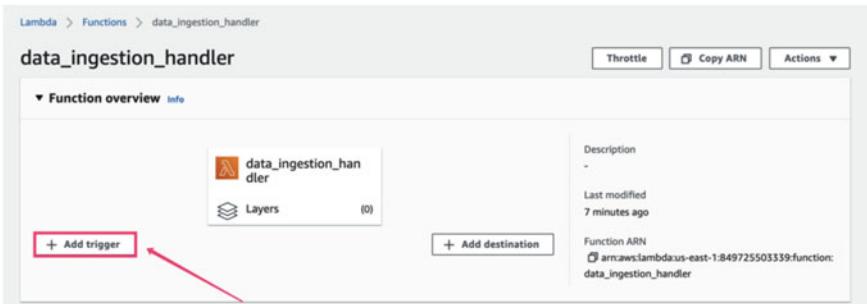


Fig. 6.9 Configuring event trigger for lambda

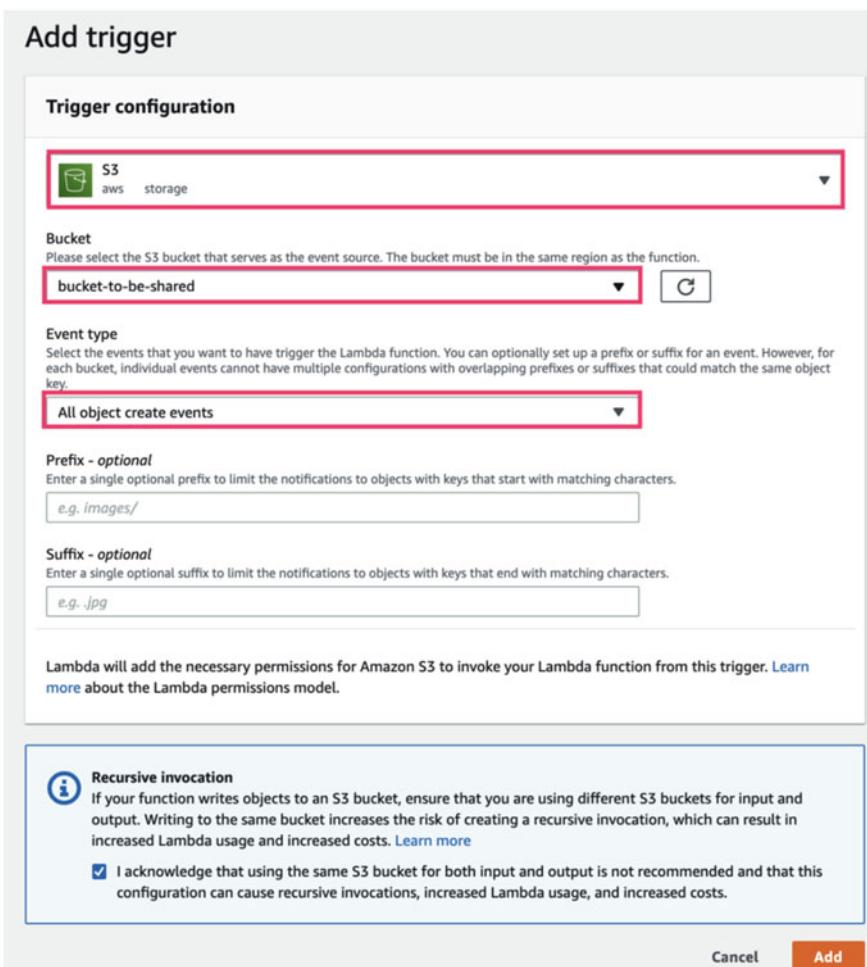


Fig. 6.10 S3 storage bucket file creation as trigger event

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "eu-west-1",
      "eventTime": "2020-04-05T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": "bucket-to-be-shared",
        "objectKey": "image1.jpg"
      }
    }
  ]
}
```

```

    "bucket": {
      "name": "bucket-to-be-shared",
      "ownerIdentity": {
        "principalId": "A3I5XTEXAMAI3E"
      },
      "arn": "arn:aws:s3:::bucket-to-be-shared"
    },
    "object": {
      "key": "uploaded_file.zip"
    }
  }
}
]
}
}

```

The next step is to add a concrete implementation for the Lambda function, which needs to accomplish the following functions.

- Parse the parameters to get the S3 storage bucket and file name.
- Download and unzip the file, get the CSV file.
- Parse and verify the file and write the records to the database.

The implementation of the code can be referred to the official case and will not be described in detail here.

3. Database Selection

One of the biggest advantages of MySQL is that it supports transactions and complex multi-table join calculations, but it is not used in our scenarios. As a row-oriented database, MySQL is not good at dealing with large data volume, and, as a stateful service, it is also more complicated and difficult to achieve elastic scaling. Here, we can use a managed database service such as Amazon DynamoDB (hereinafter referred to as DDB) instead, which is a distributed key-value database provided by AWS and can scale in million seconds. Also, DDB is a fully managed service, with no preconfigured and managed servers and no software to install or maintain.

Thus, the whole architecture is simplified in Fig. 6.11. The API/FTP and NFS building process was avoided by cross-account sharing of S3. Choosing AWS Lambda for tasks achieves better isolation between. Triggering background tasks through S3 events avoids the introduction of message queues and solves the problem of low resource utilization. Using managed DDB service eliminates the

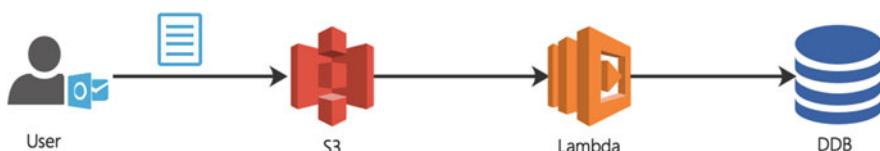


Fig. 6.11 Ingesting data to DDB

process of building and maintaining MySQL and obtains high performance and elastic scaling under large amount of data scenarios.

Next, we highlight some considerations during the Lambda implementation.

(a) Memory Allocation Model

For the Lambda runtime system resources, we only need to focus on memory, which can range from 128 MB to 10 GB. This is actually a simplified configuration; in fact, the number of CPUs and network bandwidth allocated to Lambda runtime will increase linearly with the size of the memory configuration. A Lambda instance configured with 1280 MB of memory can potentially run 10 times faster and download files 10 times faster than an instance configured with 128 MB of memory. Of course the actual difference will vary depending on the type of task and the thread model of the code. Another point is that Lambda is billed based on the product of memory size and execution time, which means that, although the lower memory configuration will make the price of execution time per unit cheaper, it will also make the total execution longer, and there may be cases where the higher configuration is more cost-efficient due to the shorter execution time.

(b) The Size of /tmp Directory Is Limited to 512 MB

/tmp is the directory used to store the downloaded S3 files. In our case, the size of the single file provided by the customer was about 800 MB, which obviously exceeded the limit. So we asked the customer to compress the file before uploading, and the ZIP file was about 50 MB, which not only met Lambda's requirements but also made the file upload and download faster. Lambda also performs a size check before downloading files from S3. If the file size exceeds 512 MB, Lambda will send an error email and exit execution. If the file size has to exceed the size limit, other solutions should be considered, such as Amazon EFS (Amazon Elastic File System) or downloading the file directly to memory.

(c) File Processing

For file processing, we can do reading while decompressing. On the one hand, it can improve the processing efficiency, and, on the other hand, it can reduce the memory consumption. The records after parsing are sent to the database in a parallel, and this process is actually a simple producer-consumer model, as shown in Fig. 6.12. Since only one file is processed at the same time, and it is read directly from the local disk, it is fast, so we only set up one producer. Also, for the consumer, which needs to send data to the remote database and wait for the response, the number can be set more. The exact number also needs to be determined based on Lambda's configuration, remote database load, and multi-round performance test results.

(d) Timeout

In order to achieve better load balancing, the maximum time for a single Lambda run is limited to 15 min, and once this time is exceeded, the task is terminated. Adequate testing of the program performance is also an essential part.

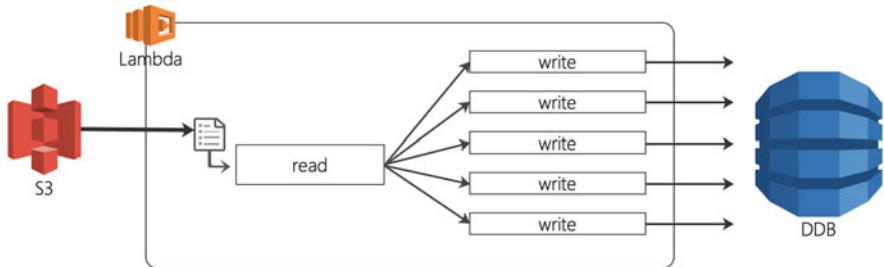


Fig. 6.12 Parallel data ingestion via lambda

(e) Concurrency

By default, the maximum number of Lambda that can be executed in parallel is 1000; i.e., if the user uploads 1000 files at the same time, it is possible that 1000 Lambda executions will be triggered in parallel. Although it is not a big deal for Lambda service, it may cause some performance pressure on DDB. Therefore, limiting the concurrency would be a better idea.

(f) Instances Reuse

Lambda is a stateless service, but, for performance reasons, its instances are not used up and destroyed but will be reused after a period of time, and resource leaks will directly affect the next execution. In addition to the /tmp directory size limit of 512 MB we mentioned earlier, Lambda also has limits on the number of file descriptors and threads, and we need to clean up these resources in time after using them.

6.3.3 Log Collection and Processing

Logging service is a very important feature in observability. With log, we can monitor our services and do troubleshooting. Log also records user behavior and provides important basic information for later analysis and report. For services under traditional architecture, multiple products are usually combined together as a solution, for example, Kafka and ELK. This part will be covered in detail in Chap. 7. For Serverless applications, cloud providers are providing services to collect logs, such as CloudWatch. The specific log collection architecture is shown in Fig. 6.13. Most AWS services integrate with CloudWatch by default, including Lambda.

For example, for a Lambda named test_log_function, when the logs are output in the code, we can see the logs in CloudWatch as shown in Fig. 6.14.

(1) is the log group name, which is automatically created by CloudWatch when Lambda is run for the first time, named as /aws/lambda/< function name>; the corresponding group name for test_log_function is /aws/lambda/test_log_function.

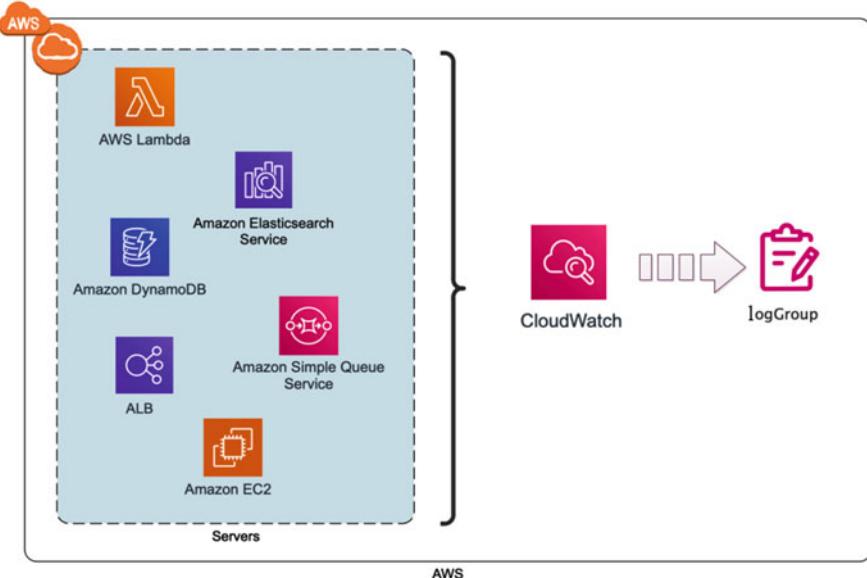


Fig. 6.13 Log collection architecture

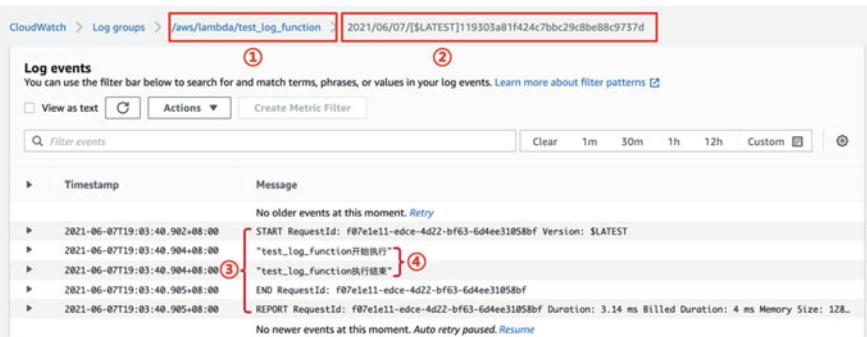


Fig. 6.14 CloudWatch logs of lambda

(2) is the log record. Each time Lambda runs, an instance is started, and each instance has one and only one record. All logs generated by that instance during its run are written to the same record. Since Lambda instances are possibly reused, logs generated by the same instance are displayed in this record.

(3) is the log generated when this Lambda is triggered, which is divided into two categories. The first category is the logs typed in the user code; see section (4). The second category is the Lambda system-level logs, such as memory and execution time statistics.

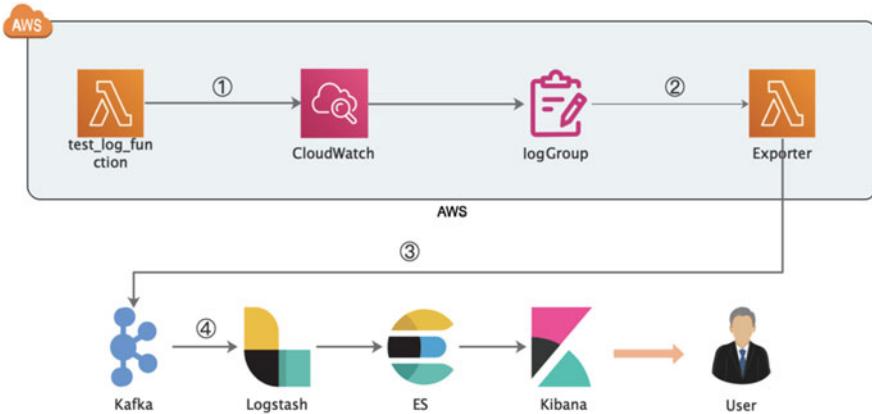


Fig. 6.15 Collecting logs and metrics from CloudWatch to ELK

While CloudWatch enables log collection and visualization out of the box; however, it is impossible to ignore the fact that CloudWatch can only be used for AWS services and requires users to be authenticated by AWS. For systems that use Serverless for only some of their services, maintaining two logging and monitoring systems at the same time is not an easy task. On the other hand, the log-based analysis and visualization features provided by CloudWatch are quite basic compared to well-known visualization products, such as Kibana.

We also encountered the above problem in our case. Our solution for this is to import the log data into our existing ELK system for unified configuration and management. Next, we will describe how to collect CloudWatch log data into ELK.

As we all know, ELK uses the Logstash service to collect and transform data and send it to Elasticsearch, and Logstash uses plugins to grab data from a number of different data sources, including CloudWatch. However, in order for Logstash to support CloudWatch, it has to create the relevant AWS user and access policies, generate access keys for them, and write them in the configuration file, which is tedious to configure. It is also important to note that the Logstash service itself is stateful and requires built-in files to record the location of records currently being processed in case of possible crashes and restarts. This imposes limitations on the multi-node deployment and scalability of the Logstash service. The more recommended approach is to introduce Kafka message queues, which can ensure the reliability and stability of log transfer data, and, with Kafka's persistence and buffering mechanism, it will not lead to data loss even if Logstash fails. This is also the architecture we choose.

As we mentioned in the previous section, AWS Lambda supports event triggering, and CloudWatch's log generation is also an event and provides direct integration with Lambda. Our architecture can be organized in the way shown in Fig. 6.15.

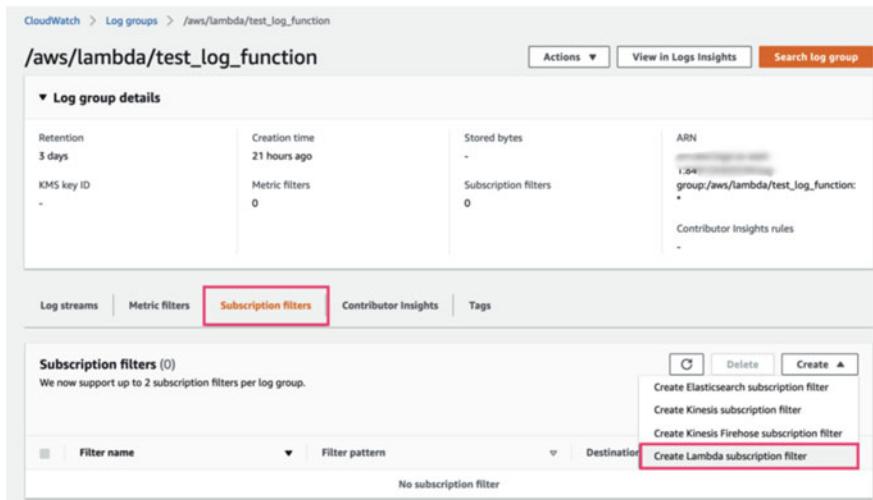


Fig. 6.16 Configuring subscription filters

In step (1), the `test_log_function` Lambda sends the logs to CloudWatch and stores them in the corresponding log group. This process is automatic and does not require any configuration by the developer.

In steps (2) and (3), the arrival of data in the log group triggers the execution of the downstream Exporter Lambda. The Exporter Lambda is then responsible for pushing the received data to the remote Kafka-specified topic. In this process, the developer needs to do two things, as follows.

First, register the Lambda listener event for the log group in CloudWatch.

As shown in Fig. 6.16, the developer selects the Subscription filters tab in the Logs group, clicks the Create button, and selects Lambda, which then jumps to the configuration page. Here, you can select the Lambda you want to register; in this case, we select Exporter as the data recipient and specify which formats of logs will be sent to the Exporter Lambda by configuring the match pattern of the logs; if not filled in, it means that all logs will be sent to the downstream Lambda. In the end, click “Start streaming” button to complete the configuration, as shown in Fig. 6.17.

The second to do is to implement Exporter Lambda.

When CloudWatch triggers Exporter Lambda, it sends data in the following format as parameters. The more important properties are the name of the logGroup and the timestamp and message in the logEvents.

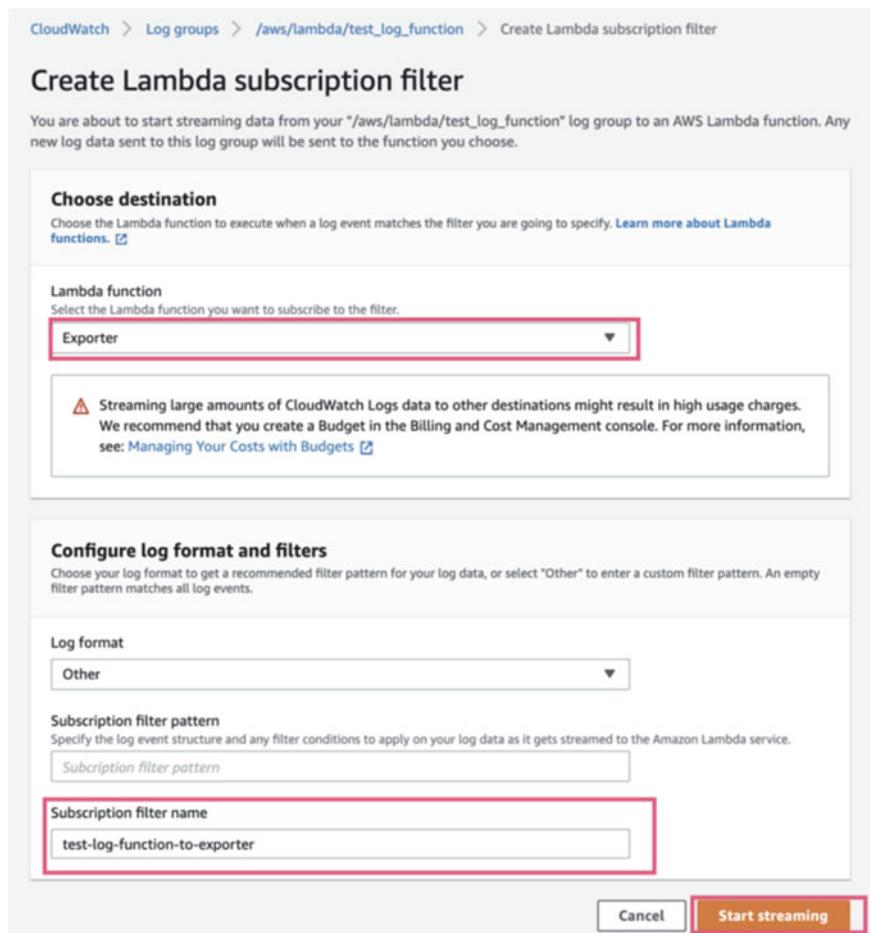


Fig. 6.17 Configuring exporter lambda

```
{
  "awslogs": {
    "data": "xxxx" //Base64 string
  }
}

// value of data field after decoding
{
  "messageType": "DATA_MESSAGE",
  "owner": "123456789012",
  "logGroup": "/aws/lambda/test_log_function",
  "logStream": "2021/05/10[$LATEST]94fa867e5374431291a7fc14e2f56ae7",
  "subscriptionFilters": [
    "test_log_function_to_exporter"
  ],
  "logEvents": [
    {
      "id": "34622316099697884706540976068822859012661220141643892546",
      "timestamp": 1552518348220,
      "message": "test_log_function start running"
    }
  ]
}
```

In the Exporter Lambda implementation, the above parameters need to be parsed to extract the log group name, timestamp, and log content and send these to Kafka with the following handler logic. Note that we have omitted details such as error handling, configuration reading, etc. for presentation convenience.

```
func Handler(ctx context.Context, events events.CloudwatchLogsEvent) error {
    // Parse to get structured data
    logData, err := events.AWSLogs.Parse()
    if err != nil {
        return err
    }

    // Create a new Kafka producer instance using sarama
    producer, _ := sarama.NewSyncProducer([]string{"your_kafka_address"}, sarama.NewConfig())

    // Get the logGroup value
    logGroup := logData.LogGroup

    // Iterate through the messages of all events to build Kafka messages
    var msgs []*sarama.ProducerMessage
    for _, event := range logData.LogEvents {
        m := &Message{
            LogGroup: logGroup,
            Timestamp: event,
            Message: event,
            Message,
        }
        msg := & sarama.ProducerMessage{
            Topic: "your_kafka_topic_name",
            Value: m,
        }
        msgs = append(msgs, msg)
    }

    // Send messages to Kafka
    return producer.SendMessages(msgs)
}
```

In step (4), Logstash consumes the data in the topic and pushes it to Elasticsearch's index.

It is important to note here that we have two types of logs: debug logs and request logs. The debug log is used to record the behavior of the program in terms of program output behavior, while the request log is used to record the behavior of the user in terms of a single interface call.

Since all logs generated by the program are treated as normal strings in CloudWatch, in order to distinguish between different log types, we can output the logs as JSON and add a flag field to the JSON structure to indicate its type, such as adding a field named `log_flag`.

For request logs, the value of this field can be set to `REQUEST_LOG_RECORD_KEYWORD` so that Logstash can push the log to a different Elasticsearch index depending on the log type when parsing it.

The following code shows the processing when the program outputs the request log.

```

func main() {
    lambda.Start(Log(api.List))
}

func Log(forward orderContext.HandleAlbRequestFunc) orderContext.HandleAlbRequestFunc {
    return func(ctx context.Context, req events(ALBTARGETGroupRequest) (events.ALBTARGETGroupResponse, error) {
        // Build the request log structure
        var record = & RequestLogRecord{}

        // Output the request log before the request returns
        defer func() {
            fmt.Println(record.String())
        }

        // When a panic occurs in the program, log the panic call stack information
        defer func() {
            if err := recover(); err != nil {
                record.PanicError = fmt.Sprintf(err, "\n", string(debug.Stack()))
                panic(err)
            }
        }
        return invokeHandler(ctx, req, forward, record)
    })
}

func invokeHandler(ctx context.Context, req events(ALBTARGETGroupRequest, handler orderContext.HandleAlbRequestFunc, record * RequestLogRecord) (events.ALBTARGETGroupResponse, error) {
    // Set a flag for the request log
    record.LogFlag = requestLogRecordKeyword

    // Record the execution time of the user function
    start := time.Now()
    res, err := handler(ctx, req)
    record.Duration = (time.Now().UnixNano() - start.UnixNano()) / int64(time.Millisecond)

    // Record the error message returned by the user function
    if err != nil {
        record.ErrorMsg = err.Error()
        if stackError, ok := err.(errors.ErrorTracer); ok {
            record.ErrorStackTrace = stackError.Trace()
        }
    }

    // Set other properties
    // ....
    return res,
err}

```

The following is the configuration file for Logstash:

```

input {
    kafka {
        bootstrap_servers => "your_kafka_address"
        topics => ["test_log_function_to_exporter"]
        # ...other properties
    }
}

filter {
    date {
        match => [ "timestamp", "UNIX_MS" ]
        remove_field => [ "timestamp" ]
    }

    if [logGroup] in [
        "/aws/lambda/test_log_function_to_exporter"
    ]{
        json {
            source => "message"
            skip_on_invalid_json => true
        }

        if [log_flag] == "REQUEST_LOG_RECORD_KEYWORD" {
            mutate {
                add_field => { "[@metadata][logtype]" => "request_log" }
                remove_field => [ "log_flag", "message" ]
            }
        } else {
            mutate {
                add_field => { "[@metadata][logtype]" => "debug_log" }
            }
        }
    }
}

output {
    if [@metadata][logtype] == "request_log" {
        elasticsearch {
            hosts => [ "your_elastic_search_addressable" ]
            index => "test-log-function-to-exporter-request-log-%{+YYYY.MM}"
            # ... other properties
        }
    } else if [@metadata][logtype] == "debug_log" {
        elasticsearch {
            hosts => [ "your_elastic_search_addressable" ]
            index => "test-log-function-to-exporter-debug-log-%{+YYYY.MM}"
            # ... other properties
        }
    }
}

```

By this point, users can query Lambda's log data through Kibana.

The debug log shown in Fig. 6.18 contains the time, log group name, debug level, and log message.



Fig. 6.18 Debug log example

As shown in Fig. 6.19, the request log contains details such as timestamp, processing time, log group name, response message, request message, and user ID.

Next, we highlight some of the problems that we may encounter during this process.

Problem 1: Traffic Explosion

A traffic explosion is when Lambda is triggered indefinitely and repeatedly due to a misconfiguration. This is not usually a problem when Lambda is handling external events. However, in this example, Exporter Lambda is triggered by a CloudWatch log group event. Whenever a Lambda executes and outputs logs, Exporter Lambda is triggered, and Exporter Lambda itself generates logs. Once we configure the Lambda to listen to its log group by mistake, a triggering loop is created, at which point the number of Lambda executions grows exponentially and quickly exhausts the AWS concurrency limit.

To prevent this from happening, in addition to carefully reviewing the configuration when deploying Lambda, you can also reduce the impact on other services by limiting the maximum number of concurrency of Lambda and setting Lambda metrics monitors. For situations that have already occurred, the concurrency value can be set to 0 to force termination of execution.

Time ..	logGroup	duration	http_status_code	network_id
May 15th 2021, 01:00:43.114	/aws/Lambda/Lambda-PRO- XXXXXXXXXX	1,967	200	169843
Table	JSON		View surrounding documents	View single document

Fig. 6.19 Request log example

Problem 2: Single Log Size Limitation

Lambda has a size limit when exporting logs to CloudWatch. If the size of a single log exceeds 256 KB, then it will be cut into multiple log records. In our request log implementation, the program needs to serialize the request log into a string for output to CloudWatch and then perform JSON parsing in Logstash. Compared to debug logs, request logs are generally larger and can be even larger if the request body and response body are involved. Once it exceeds 256 KB, a single JSON record is cut into two records, and Logstash cannot parse it into a request log.

To avoid this, you can check the log when it is output, and, if it exceeds the size limit, split it into two records, or truncate some unnecessary attributes and keep only the critical information. Compression is also possible, but it makes the logs in CloudWatch unreadable and not recommended.

6.4 Summary of This Chapter

This chapter focuses on Serverless. Section 6.1 briefly describes the basic concept and development history of Serverless and elaborates on its advantages and shortcomings. Section 6.2 summarizes several typical application scenarios based on the characteristics of Serverless. Section 6.3 introduces two AWS Lambda-based Serverless landing cases with our experience. Section 6.3 introduces two AWS

Lambda-based Serverless implementation cases based on our experience and discusses the key technical points in the implementation in detail.

It is important to note that Serverless allows developers to focus more on business logic and rapid deployment, and it is better suited to solve a certain type of problem. It is also important to realize that the goal of Serverless is not to completely replace existing architectures but to provide developers with another option in certain scenarios.

Chapter 7

Service Observability



When business systems adopt microservices architecture and migrate to the cloud, we face many challenges, especially how to view the status of each individual service of the application, quickly locate and resolve online issues, and monitor the call relationships between services. In the cloud-native world, observability is one of the categories used to address these challenges. This chapter will introduce the definition and application of service observability, the difference and relation between observability and monitoring, the current state of the observability community, and our team's practice on observability adoption.

7.1 What Is Observability

Observability is not a new term; it comes from the field of control theory, which is a measure of how well the internal states of a system can be inferred from knowledge of its external outputs. It is widely used in electrical and mechanical engineering and other industrial fields. The concept was first introduced to the microservices and cloud-native domains in 2017 and subsequently occupied an important subgroup in the CNCF Landscape. As of April 2021, there are 103 projects under this group, and three of the top 10 projects that successfully graduated from CNCF belong to this group, namely, Prometheus, Fluentd, and Jaeger. In this section, we will dissect the definition and character of observability.

7.1.1 *Definition of Observability*

After observability was introduced into the microservices and cloud-native domains, there is no unified, industry-accepted definition, and I prefer the explanation given by Cristian Klein, the Senior Cloud Architect at Elastisys.

Taking measurements of your infrastructure, platform, and application to understand how it's doing. As Peter Drucker used to say: "If you can't measure it, you can't manage it."

From this explanation, it is clear that the ultimate goal of introducing observability is to better manage the system. In a complex microservice system, observability can help us understand and measure the operational state of the system, determine whether there is room for optimization, and locate problems such as the following.

- What is the status of each service and is it processing requests as expected?
- Why did the request fail?
- What services do customer requests go through, and are there performance bottlenecks in the invocation chain?

As for how to make a system observable, the key lies in how to provide data that reflects the operational state of the system, and how to collect and display this data and properly alert when the system is not working as expected. We will elaborate on these elements later.

7.1.2 *Three Pillars of Observability*

As mentioned in Sect. 7.1.1, observability is based on how to provide data that reflects the operational state of a system. Peter Bourgon, after attending 2017 Distributed Tracing Summit, published the famous article "Metrics, tracing, and logging," which succinctly described three pillars of observability in distributed systems and the connections and differences between them, namely, Metrics, Logging, and Tracing, as shown in Fig. 7.1.

Each of these three types of data has its focus and needs to be combined to become an observable data source.

- Metrics: A logical gauge, counter, or histogram over a period that is atomic and cumulative. For example, how much memory was used for a particular request, how many requests were processed by a particular service in the past 1 h, etc. As can be seen in Fig. 7.1, metric data takes up the lowest amount of volume and can be transferred and stored efficiently.
- Logging: The records of system events that are discrete and immutable. For example, it records error messages when a service goes wrong, records information about how the system handled a particular request, etc. As you can see in Fig. 7.1, logs take up the highest volume because they can carry a lot of information to help us debug problems.
- Tracing: Information within the scope of a single request, all data, and metadata information within the lifecycle of a particular request is bound to a single transaction. For example, which services or modules in the system a request went through, and what was the processing status (errors, latency, etc.) on those services or modules. As you can see in Fig. 7.1, the volume consumed by tracing

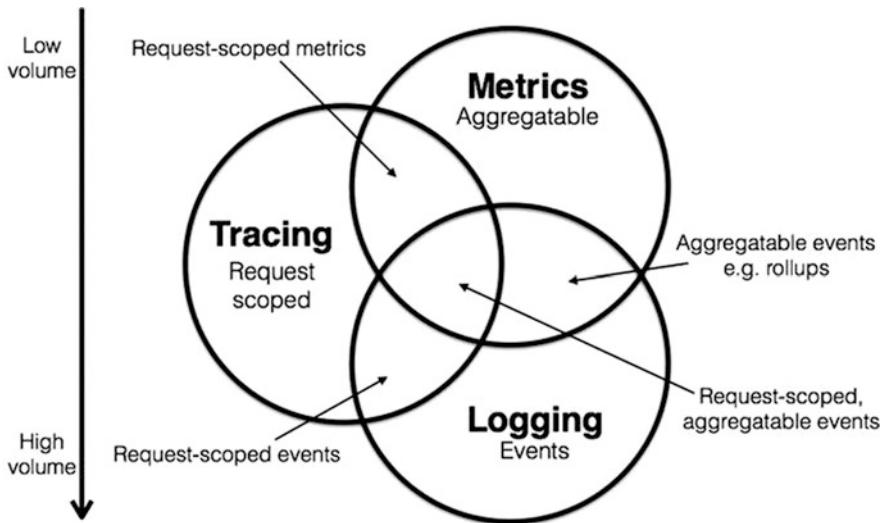


Fig. 7.1 Three pillars of observability in distributed systems (from <https://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html>)

data is somewhere between metrics and logs, and its main role is to link information from various services or modules to help us locate problems quickly.

With these three types of data, the troubleshooting process for a typical problem in a distributed, microservices-based system is rough as follows:

1. First, discover anomalies (usually tracing or log data) by predefined alerts.
2. When anomalies are found, the invocation chain information recorded by the distributed tracing system is used to locate the problematic service (tracing data).
3. Query and analyze the logs of the service with the problem to find the key error messages, and then locate the code, configuration, etc. that caused the problem (log data).

7.1.3 *The Difference and Relation Between Observability and Monitoring*

Many people think that the so-called observability is just the same as traditional monitoring, but, in my opinion, the two are close in meaning, but there are still differences.

Simply put, monitoring is a subset and key function of observability, and the two complement each other. A system cannot be monitored without some level of observability. Monitoring can tell us how the system is performing overall and

when something is going wrong, while observability can further help find the cause of the problem.

Traditional monitoring is usually the responsibility of operations engineers, and, at first, monitoring was even used only to check if the system was down. According to Google's Site Reliability Engineer's Handbook (SRE Book), *your monitoring system should address two questions: what's broken, and why?*

For a simple monolithic application, operations engineers monitoring network, host, and application logs are basically enough to answer these two questions. However, as the application becomes more complex, we usually transform it into a microservice architecture, and it becomes increasingly difficult to solve these two problems by relying only on operations engineers. We need developers to get involved and consider how to expose the information inside the application from the system design and development phase, including the network, hosts (or containers), infrastructure, and the service itself, to make the whole application observable. We usually expose the internal state of the service (i.e., the metrics, logs, and trace data introduced in Sect. 7.1.2) through instrumentation. It can be said that observability follows the DevOps concept to some extent—combined software development with operations.

With observable data, we can monitor the application efficiently.

- The operational status of the system as a whole or a particular service is presented through a series of Dashboards, which are usually a collection of Visualizations.
- When the system is abnormal, the predefined alert rules are used to notify the relevant personnel in time to deal with it. Of course, we can also use machine learning to scan data, identify abnormalities, and automatically trigger alerts, avoiding manual definition and adjustment of various alert rules.
- Use the pre-processed data (usually the tracing and logs data) to query the related contextual information to locate and resolve the problem.

Observability not only helps us to monitor the system better, but it also has the following functions.

- Optimize systems: Analyze data to better understand the system and customer usage, so that you can effectively optimize the system (optimize invocation chains, business processes, etc.).
- Failure prediction: Failures always cause damage, so the ideal situation is to be able to avoid them. We can do Proactive Monitoring by analyzing some precursor information from previous failures.

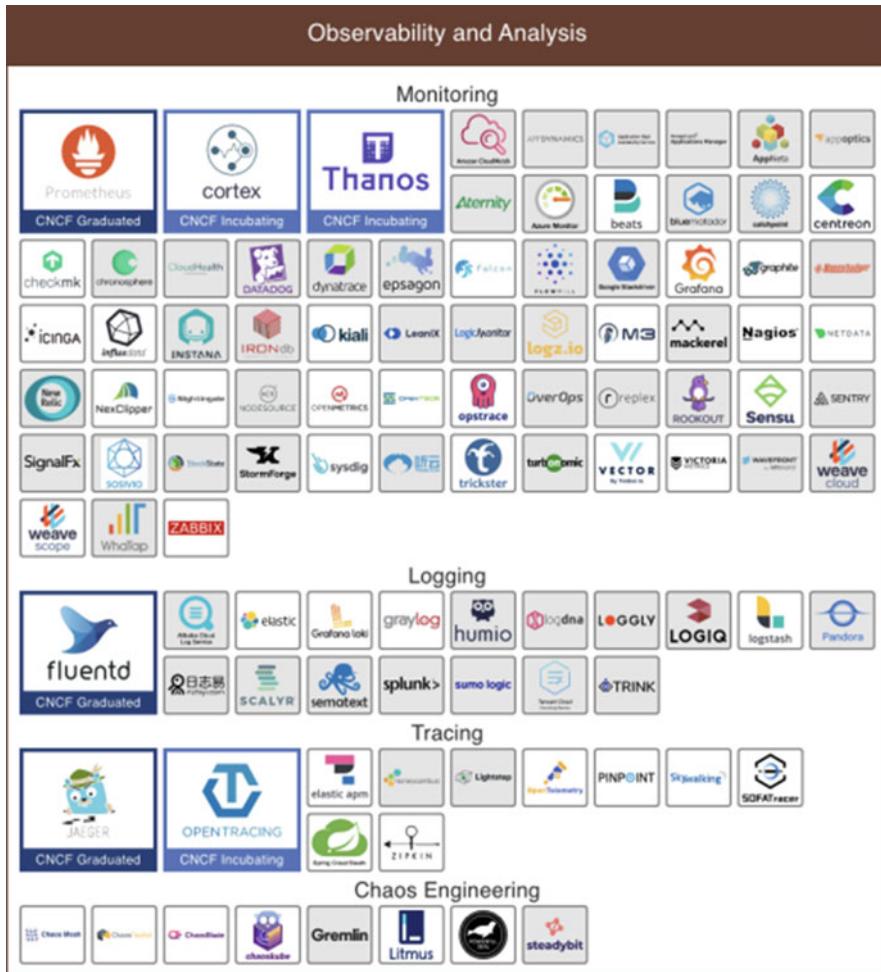


Fig. 7.2 CNCF landscape for observability and analysis (from <https://landscape.cncf.io>)

7.1.4 Community Products and Technology Selection

As mentioned earlier, since the emergence of the concept of observability in the cloud-native domain, there are plenty of products emerging, showing a blossoming landscape, as shown in Fig. 7.2.

As we can see from Fig. 7.2, the products at this stage essentially have their own focus, with most of them focusing on one of the three pillars of observability, and there is a lack of a grand unified product that can address and relate the three pillars simultaneously.

- Focus on metrics: Prometheus, InfluxDB, Cortex, Zabbix, Nagios, OpenCensus, etc.
- Focus on logging: ELK, Fluentd, Splunk, Loggly, etc.
- Focus on tracing: Jaeger, Zipkin, SkyWalking, OpenTracing, OpenCensus, etc.

When designing the observability solution for our business application, we chose multiple products (Prometheus, InfluxDB, Cortex, ELK, and Jaeger) to accomplish the collection and usage of the three pillars of data, and the detailed adoption solution will be introduced in the later sections.

Of course, there are some problems when multiple products/projects are in mixed-use, such as huge maintenance effort and non-interoperable data. The lack of data interoperability is reflected in the fact that the data requested by customers is stored in each product's own system and cannot be fully utilized.

The industry has already been aware of this problem, OpenTelemetry under CNCF aims to unify metrics, logs, and traces to achieve observability data interoperability.

OpenTelemetry is a collection of tools, APIs, and SDKs. Use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help you analyze your software's performance and behavior.

OpenTelemetry is positioned as an observability infrastructure, solving the problem of data specification and collection, but the subsequent analysis and presentation need to be done with other tools, and there is no unified platform that can do all the work yet. We will closely monitor the progress of this project and optimize our observability solution in due course.

7.2 Logging Solutions Under Cloud-Native

During the construction and development of most business systems, logs, the output behind the system operations, describe the behavior and state of the system and are the cornerstone for developers and operators to observe and analyze the system. In the early stages of application construction, logs may exist only for variable printing and logic debugging purposes, and their content may only be stand-alone records. As systems become complex and the interactions between services increase, it is more important that the log content needs to meet the precision, normalization, and ease of observation. This section gives a brief overview of the cloud-native logging solution that we have been thinking about, developing, and iterating over the years.

7.2.1 *Log Classification and Design*

To standardize and normalize the log for achieving maintainability, scalability, and observability, we divide the logs recorded by the application in different usage

scenarios into Debug Logs and Request Logs and developed related tool libraries and middleware to reduce the burden of developers.

1. Debug Log

Debug Log is the most common and basic type of logging. It is usually implemented by developers who embed logging code in their code logic to record the behavior and state of the system that is or has occurred. To be able to provide good readability and maintainability, we first defined a preliminary logging specification.

- On the log marker, the timestamp of the output log is to be recorded for easy access and traceability of what happened at what moment and in what order.
- On the log classification, the current log level needs to be stated. Generally, the levels are Debug, Info, Warn, Error, Fatal, and Panic, where Panic is not recommended in non-extreme cases.
- In terms of log content, do not simply print variables or structures, but add basic descriptions and relevant auxiliary information to illustrate and guide the purpose and significance of the current log content, to avoid wasting human and material resources by having meaningless logs.

With the above specification, we get a logging structure that can categorize logs by time and level. The common Golang-based logging libraries, such as Logrus, Zap, Zerolog, log15, etc., can print logs that meet the above specification on demand and provide a variety of features with excellent performance. With the development of technology iteration and rapid changes, more excellent log libraries may appear later for us to use. To enable better switching and iteration of various logging libraries, we encapsulated a new logging tool library on top of the existing logging libraries, implemented a set of common function definitions according to our business requirements and development needs, and made the underlying logging libraries transparent. In this way, when we update and iterate the underlying log library in the future, we can minimize the coding impact on the application service and reduce the maintenance cost. In addition, to make the logging structure easier to collect, parse, organize, and analyze, the logging library implements the following functions on top of meeting the above three requirements.

- Dynamically configurable and adjustable output format of the timestamp, default is RFC3339.
- Dynamically configurable and adjustable output level, default is Info.
- Dynamically configurable and adjustable output format of the log, default is JSON format and can be changed to Text.
- Extracting trace information from the Context after the application integrated with Tracing for easy query and reference among related logs.
- Customized key-value log content to facilitate the custom expansion of log descriptions.
- Record the size of log data to analyze the scale and review the reasonableness of the logs.

For example, we can use the log library in the following way:

```
// Example of using Debug Log
func logExample() {
    // Initialize Tracing and inject trace information into Context
    tracing.InitDefaultTracer("example")
    sp := tracing.StartSpan("example_span")
    sp.Finish()
    ctx := opentracing.ContextWithSpan(context.Background(), sp)

    // Configurable log output format and time output format, default is JSON and RFC3339
    log.ConfigFormat(log.JSONFormat, time.RFC3339)

    // Configurable log output level, logs lower than this level are not output, default is Info
    log.SetLevel(log.DebugLevel)

    // Customized key-value content of the log
    fields := map[string]interface{}{
        "test_key": "test_value",
    }

    // simple text with Error output level carries trace information and customized key-value content
    log.WithCtxFieldsError(ctx, fields, "test message: log example.")
}
```

The corresponding output log is as follows:

```
// log.WithCtxFieldsError(ctx, fields, "test message: log example.")
{
    "level": "error",
    "msg": "test message: log example.",
    "parent_id": "0",
    "size": 26,
    "span_id": "719a1b2fe311510a",
    "test_key": "test_value",
    "time": "2020-11-11T11:11:11Z",
    "trace_id": "719a1b2fe311510a",
    "type": "debug"
}
```

As shown above, the log prints what we want according to the content and format we set. For each content key:

- level: the level of the log output.
- time: timestamp of the log output.
- msg: the content of the log.
- size: the size of the log content.
- type: mark this log as Debug Log, which is used to process different log types during later collection.
- test_key, test_value: custom key-value.
- trace_id, span_id, parent_id: the trace information carried in the Context.

2. Request Log

In practice, we found that it is difficult to meet the needs of building application service scenarios by simply recording variables or the status of the code logic. For customer-oriented business systems, input and output are the top priority of the whole system, especially that, after the application microservitization, the requests received by the service and the content returned

to the caller are necessary for viewing and monitoring the status of the system, as well as analyzing the behavior of users and the system.

To solve this problem, we propose an interceptor middleware that records Request Logs. This middleware wraps at the beginning and end of the service request or event life cycle to extract information such as request address, request content, response content, error list, processing function, client ID, timestamp, and duration from the context in which the request occurs and prints it in JSON form. At the same time, the middleware not only records the input and output information of this service but also supports recording the input and output information from this service to other services, especially external services (such as Elasticsearch and DynamoDB), which can help us better observe and discover the behaviors and problems occurring in the service. Based on a good structure, the middleware supports services under both HTTP and gRPC protocols and is also compatible with services of Job and Message type, which facilitates the unified collection and interpretation of logs and reduces the cost of reading and maintaining logs between different services.

The implementation structure of the Request Log is shown in Fig. 7.3, through which the following functions can be implemented.

- Record the request input and response output (e.g., error code, error message, error structured content, error point stack information, the called method information and the corresponding context, the start time and duration of the request or event, the client ID that initiated the request, etc.).
- Add customizable record units and handle customized content through passed custom functions.
- Support both HTTP and gRPC Request Log messages.
- Support logging of messages for Job and Message services that are not instantly requested.
- Tracing is supported to record call chain information.

The generated Request Log calls the Debug Log directly, places the translated information in the msg field, and sets the type field to Request to distinguish it from other logs, as shown in the example below:

```
{
  "level": "info",
  "msg": "{\"TraceID\":\"7ec0301254a9af75\",\"IsOutgoing\":false,\"NetworkID\":12...",
  "size": 1517,
  "time": "2020-11-11T11:11:11Z",
  "type": "request"
}
```

Below, we list a few different types of Request Logs and describe their implementation.

(a) gRPC Request Log

For gRPC requests, we can build a unary interceptor to get the input and output of the request. Unary interceptors can be divided into unary server interceptors and unary client interceptors, which act on the gRPC server side and client side respectively. The definition of a unary server interceptor is as follows:

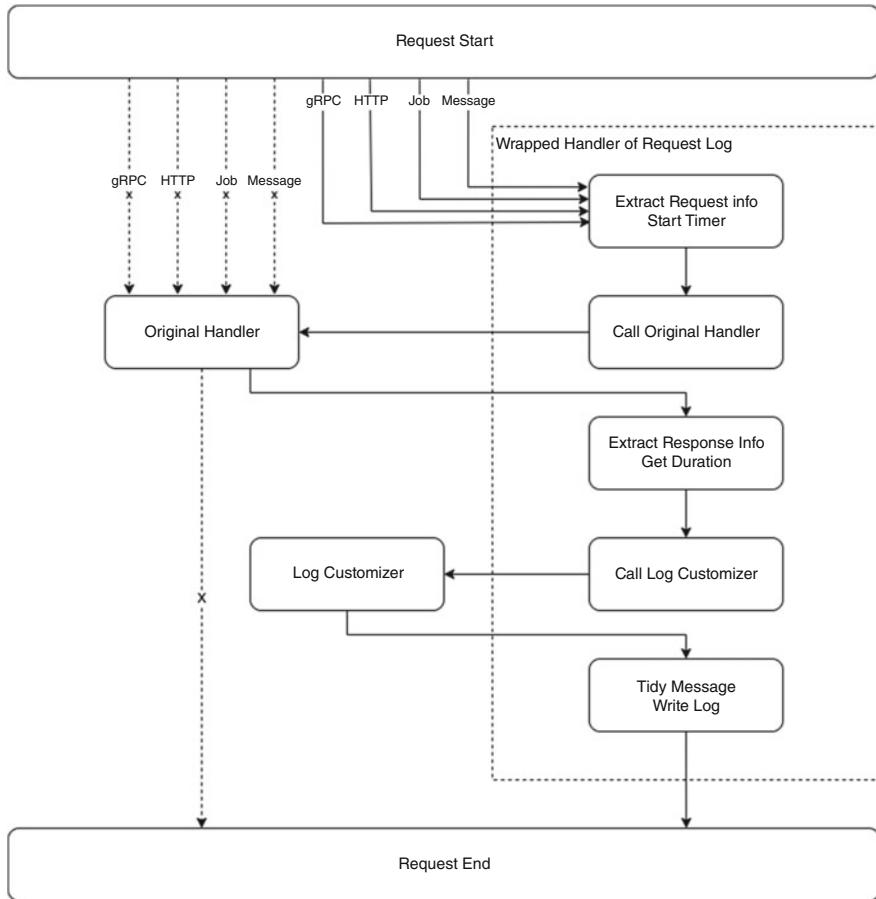


Fig. 7.3 Request log workflow

```
type UnaryServerInterceptor func(ctx context.Context, req interface{}, info *UnaryServerInfo, handler UnaryHandler) (resp interface{}, err error)
```

The unary server interceptor provides a hook that acts on the server side and can intercept gRPC requests. The function input parameter `ctx` is the context information of the request call; `req` is the content of the request; `info` is the information of all operations contained in the request, and `handler` is a wrapped implementation of a specific service handling method. The output parameters are the content of the response message `resp` and the error message `err` to be returned to the caller by the client. Based on the unary server interceptor, we implement the following method to intercept the request log:

```
// UnaryServerInterceptor, a server-side interceptor that intercepts and records server-side requests
// Pass the parameter customizeLog into the function to support external custom log handling methods to make logging more flexible
// Or just pass nil to use the default logging method
func UnaryServerInterceptor(customizeLog func(ctx context.Context, logRecord *LogRecord)) grpc.UnaryServerInterceptor {
    return func(
        ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler,
        ) (reply interface{}, err error) {
        // Initialize the Request Log structure
        var logRecord *LogRecord
        // Control the logic call before the function returns by defer
        defer func() {
            // Prevent unintended termination during the call, resulting in lost logs
            if r := recover(); r != nil {
                log.Errorf("Panic due to: %v\n%s", r, string(debug.Stack()))
                err = errors.New(fmt.Sprintf("Panic due to: %v", r))
            }
        }
        // Extract the returned results, record the duration, and output the Request Log
        finishLog(ctx, logRecord, reply, err, customizeLog)
    }
    // Extract the valid information from the request and save it to the logRecord via beginLog
    logRecord = beginLog(ctx, req, info.FullMethod, false, customizeLog)

    // Call the wrapped handler method
    return handler(ctx, req)
}
}
```

Similar to the unary server interceptor, the unary client interceptor intercepts gRPC requests on the client side. When creating a ClientConn, we can specify the interceptor as a Call Option, via the WithUnaryInterceptor method or the WithChainUnaryInterceptor method. When a unary client interceptor is set in the request, gRPC delegates all unary calls to the interceptor, which is responsible for triggering the Invoker and thus completing the processing. The definition of a unary client interceptor is as follows:

```
type UnaryServerInterceptor func(ctx context.Context, req interface{}, info *UnaryServerInfo, handler UnaryHandler) (resp interface{}, err error)
```

For the input parameters, ctx is the context information of the request call; the method is the method name of the request; req is the content of the request; the reply is the content of the response returned by the server to the client; cc is the client connection that invokes the gRPC request; invoker is the handler that completes the gRPC call; and opts contains all applicable invocation options, including default values for the client connection and each specific invocation option. Based on the unary client interceptor, we also implement logic similar to the unary server interceptor to handle Request Log messages.

```

// UnaryClientInterceptor, a unary client interceptor, implements the interception and logging of client Request Logs
// Passing the parameter customizeLog into the function can support external custom log handling methods to make logging more flexible
// Or just passing nil to use the default logging method
func UnaryClientInterceptor(customizeLog CustomizeLogFunc) grpc.UnaryClientInterceptor {
    return func(ctx context.Context, method string, req, reply interface{}, cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption)
error {
    // Initialize the Request Log structure
    var logRecord *LogRecord

    // Control the logic call before the function returns by defer
    defer func() {
        // Prevent unintended termination during the call, resulting in lost logs
        if r := recover(); r != nil {
            log.Errorf("Panic due to: %v\n% s", r, string(debug.Stack()))
            err = errors.New(fmt.Sprintf("Panic due to: %v", r))
        }
    }

    // Extract the returned results, record the duration, and output the Request Log
    finishLog(ctx, logRecord, reply, err, customizeLog)
}

    // Extract the valid information from the request and save it to the logRecord via beginLog
    logRecord = beginLog(ctx, req, method, true, customizeLog)

    // trigger the wrapped invoker method and return the error message generated by the
    invoker method
    return invoker(ctx, method, req, reply, cc, opts...)
}

}

```

(b) HTTP Request Log

Similar to gRPC, we classify HTTP requests according to server-side and client-side as well and implement Request Log extraction and logging, respectively. However, unlike the unary interceptor structure provided by gRPC, the HTTP side requires us to write our own middleware to implement the interception function.

For the HTTP server side, we construct a new HTTP handler function. This function first obtains and records the request content and then replaces the receiver of the ServeHTTP method to intercept the returned response content, and, after recording, the intercepted response content is backfilled to the original receiver and finally replaces the constructed function above with the original HTTP handler to achieve the request interception; the specific implementation logic is as follows:

```

// HTTPServerMiddleware, logging by replacing the HTTP handler function handler
// The input parameter handler is the original HTTP handler function
// The output parameter handler is the HTTP handler function containing the logging logic
func HTTPServerMiddleware(handler http.Handler) http.Handler {
    // Define the wrapper function f that replaces the original handler function
    f := func(writer http.ResponseWriter, req *http.Request) {
        // extract the context ctx from req
        ctx := req.Context()

        // Initialize the receive structure of the HTTP request
        recorder := httptest.NewRecorder()
        // Initialize the Request Log structure
        var logRecord *LogRecord

        // Control the logic call before the function returns by defer
        defer func() {
            // Prevent unintended termination during the call, resulting in lost records
            if r := recover(); r != nil {
                log.Errorf("Panic due to: %v\n%#s", r, string(debug.Stack()))
                recorder.Code = http.StatusInternalServerError
            }
        }

        // Return the HTTP headers received in the recorder that were originally returned to the writer
        for k, v := range recorder.Header() {
            writer.Header()[k] = v
        }
        writer.WriteHeader(recorder.Code)
        // Return the HTTP message received in recorder that was originally returned to writer
        WriteTo(writer)

        // Extract the return result, record the duration and output the Request Log
        finishHTTPLog(logRecord, recorder.Result(), nil)
    }

    // Extract the valid information from the request by beginHTTPLog and save it to the logRecord
    logRecord = beginHTTPLog(ctx, req, false)

    // Replace the original recipient writer with a recorder to intercept the response information returned by the server
    handler.ServeHTTP(recorder, req)
}

// Replace the handler method with the handler function f defined above
return http.HandlerFunc(f)
}

```

Unlike the HTTP server side, the HTTP client can perform a single HTTP transaction by building a RoundTripper and setting it up as the HTTP client's Transport to obtain the corresponding request and response information, thus achieving server-side middleware-like functionality. The following code demonstrates how to implement the RoundTripper interface and intercept request and response information:

```

// This structure implements the RoundTrip method, in which request and response information is obtained and recorded
type HttpClientMiddleware struct {
    // A structure that implements the RoundTripper interface, thus enabling multiple middleware to work together
    Middleware http.RoundTripper
}

// RoundTrip, which implements the RoundTripper interface to intercept and record request and response information
// The input parameter req is the request information sent by the HTTP client
// The output parameter res is the response information received by the HTTP client
// The output parameter err is the error information received by the HTTP client
func (hcm HttpClientMiddleware) RoundTrip(req *http.Request) (res *http.Response, err error) {
    // Extract the context ctx from req
    ctx := req.Context()
    // Initialize the Request Log structure
    var logRecord *LogRecord

    // Control the logic call before the function returns by defer
    defer func() {
        // Prevent unintended termination during the call, resulting in lost records
        if r := recover(); r != nil {
            log.Errorf("Panic due to: %v\n%s", r, string(debug.Stack()))
        }
    }

    // Extract the return result, record the duration and output the Request Log
    finishHTTPLog(logRecord, res, err)
}

// Extract the valid information from the request and save it to the logRecord by beginHTTPLog
logRecord = beginHTTPLog(ctx, req, true)

// Call the member Middleware's RoundTrip method to enable multiple middleware to work together
return hcm.Middleware.RoundTrip(req)
}

```

(c) Job Request Log

Job-type requests usually perform one-time or periodic tasks, which do not require a real-time return but still produce execution content and task result information. We map the execution content and output result to the request content and response content of the Request Log, respectively, to reduce the barrier and cost of log reading and maintenance by reusing the log structure and improve the efficiency of engineers' daily work.

For the self-implemented job framework by our team, we constructed middleware specifically for Job Request Log with the following code:

```

// JobRequestLogMiddleware, registered in the job processing thread pool
// Records the interception of the execution content and output results before and after the execution of the job
// The input parameter j records the execution content of the job
// The input parameter next records the next middleware to be executed
// The output parameter err returns the error message generated during this period
func JobRequestLogMiddleware(j *jobWorker.Job, next jobWorker.NextMiddlewareFunc) (err error) {
    // Initialize the Request Log structure
    var logRecord *LogRecord

    // Control the logic call before the function returns by defer
    defer func() {
        // Prevent unintended termination during the call, resulting in lost records
        if r := recover(); r != nil {
            log.Errorf("Panic due to: %v\n%v", r, string(debug.Stack()))
            err = errors.New(fmt.Sprintf("Panic due to: %v", r))
        }
    }

    // Extract the return result, record the duration and output the Request Log
    finishLog(nil, logRecord, "", err, nil)
}

// BeginJobLog to extract the valid information from the task information and save it to the logRecord
logRecord = beginJobLog(j)

// Call the next middleware method next to enable multiple middleware to work together
return next()
}

```

(d) Message Request Log

Similar to the Job Request Log, to be able to record asynchronous messages well, we map the message information and return information received by the consumer to the request and response information of the Request Log, respectively.

In the following code, we take a consumer interceptor of type proto as an example and briefly describe how to intercept and wrap the original consumer message handling methods:

```
// The number of Pods in the cluster where a service is not running properly
count(up{app="$service_name", k8s_cluster=~"$k8s_cluster"})==0
```

With the above functions, the middleware for recording Request Logs can meet the demand for Request Log functions in all current business scenarios of the system and provides enough scalability to provide a good middleware foundation for the construction of new services and troubleshooting of problems between different services.

7.2.2 Evolution of Cloud-Native Log Collection Scheme

With excellent middleware that outputs standard logs, we also needed an elegant log collection system to better categorize the logs for further observation, analysis, and monitoring. For this purpose, we chose ELK Stack, and this section describes the evolution of using this technology stack.

1. Single application period

In the period of a single application and physical machine, we collect the logs generated by the application by deploying Filebeat on the physical machine to add classification information and encapsulation by line. As shown in Fig. 7.4, Filebeat as the producer is responsible for passing the encapsulated messages to the message queue Kafka cluster, after which Logstash as the consumer gets the log messages from the Kafka cluster; classifies, cleanses, parses, and processes them; and then passes them to the Elasticsearch cluster for storage. The logs generated by the system can then be viewed through Kibana, and dashboards can be built on top of that to further visualize the log information.

2. Early solution for service containerization: Sidecar

With the containerization of services, the running environment between services is relatively independent and no longer constrained by physical machines, so we then initially tried to integrate Filebeat in the Pod of services in the form of Sidecar.

As shown in Fig. 7.5, Filebeat acts as a Sidecar in the Pod to collect the log files under the corresponding path and send them to the Kafka cluster. The configuration of Filebeat in this scenario is very similar to the physical machine period, and the changes are relatively small as a transition solution. However, the resource consumption of Sidecar runtime can affect the entire Pod, which in turn affects the performance of the service. We once encountered a situation in which the memory resources of the whole Pod were severely consumed due to the short time and high frequency of log output, and the business services could not continue to serve, thus causing the Pod to restart automatically. In this case, we could only temporarily increase the amount of Pods and adjust the resource allocation in the Pods, provisionally relieving the load in the high-throughput phase after expanding the configuration to several times the original one but still losing part of the requests and corresponding logs between the time the problem occurred and the completion of the adjustment. Therefore, to fundamentally circumvent this problem, we propose to adopt the DaemonSet scheme on this basis.

3. Service containerization stabilization solution: DaemonSet

We redirect the logs (Debug Logs, Request Logs, third-party logs) from the console to the storage location of the node and later make Filebeat in the node monitor and collect the path where the logs are located. This solution effectively separates the operation of Filebeat from the application's container and Pod, avoiding the resource loss to the application caused by using Sidecar in Pod. Filebeat, started in DaemonSet mode, initially processes the collected logs and, after adding and encapsulating information about the Kubernetes host environment, passes the information to the corresponding Kafka cluster, as shown in Fig. 7.6.

For the specific configuration of Filebeat, since there is a lot of related literature, we will not go into it here. We will only describe it for its input in Docker mode, with the following code:

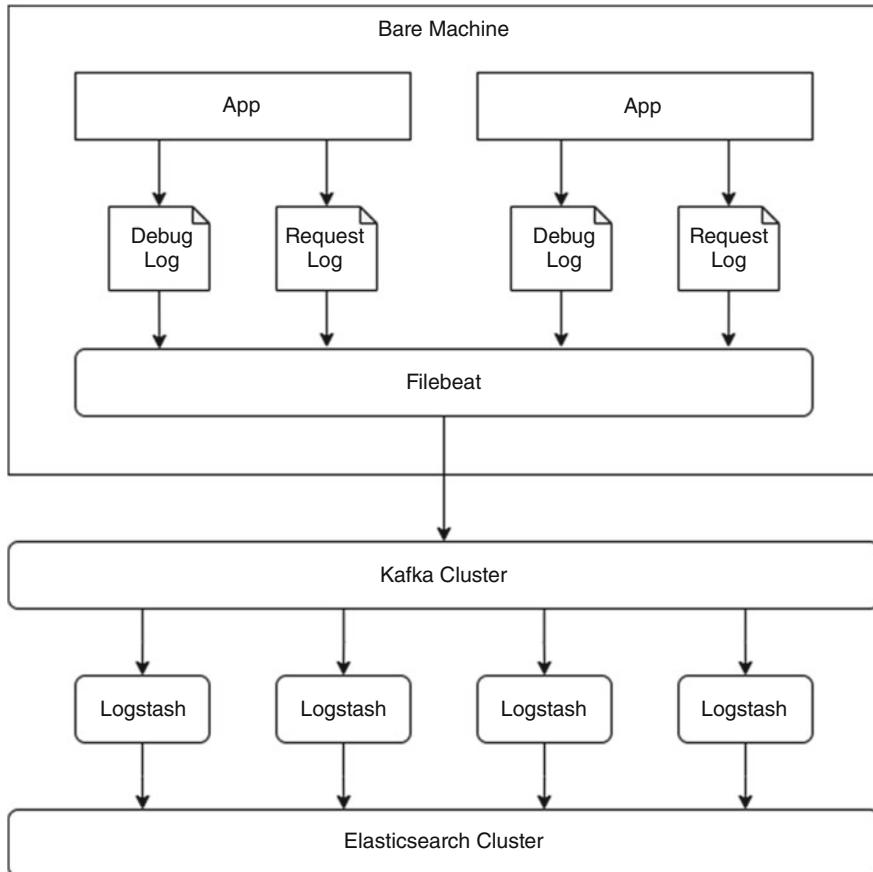


Fig. 7.4 Log collection for bare machine

```

- combine_partial: true # Enable shard log integration
containers:
ids:
- '*'
path: /var/lib/docker/containers
stream: all
exclude_files: # Declare targets that do not need to collect logs
- filebeat_*
\log
-logstash-* \log
fields: # with custom field information
dc: AWS-DC-1
k8s_cluster: EKS-PRD-K8S
processors:
- add_kubernetes_metadata: # append collected Kubernetes metadata to the log content
  default_matchers.enabled: false # Disable default matchers for index creation
in_cluster: true # Run as a Pod inside a Kubernetes cluster
include_annotations:
# Attach Kubernetes meta information when
log-collection flag is detected on log-generating Pods - log-collection
matchers: # Custom index creation matchers
- logs_path: # Use log path information stored in the log.file.path field to create indexes
logs_path: /var/lib/docker/containers
- drop_event: # Drop log messages
when:
not:
has_fields when it detects that the Pod generating the log does not carry the log-collection flag
:
  - kubernetes.annotations.log-collection
- drop_fields: # Drop useless fields
fields:
- input
- prospector
- kubernetes.labels
- beat.hostname
symlinks: true # Support Get log information from linked files
type: docker # Mark inputs fields as type docker

```

After Filebeat passes log messages to the Kafka cluster, Logstash, as the consumer of the Kafka cluster, listens to the messages in the registered topic and consumes them. When Logstash receives a new message, it distinguishes whether it is a Debug Log, a Request Log, or a third-party log based on the type in the log message and parses it differently depending on the log type. Thanks to the Kubernetes meta information and other host information that Filebeat adds when collecting logs, we can dispatch logs into different Elasticsearch indexes based on container names (at this point, we combine third-party logs with debug logs in the same Elasticsearch index for easy log query) in the following form:

prefix-%{kubernetes.container.name}-%{log.type}-log-%{+YYYY.ww}
Example: fw-ui-order-request-log-2021.01

The specific Logstash filter code and output configuration information is as follows:

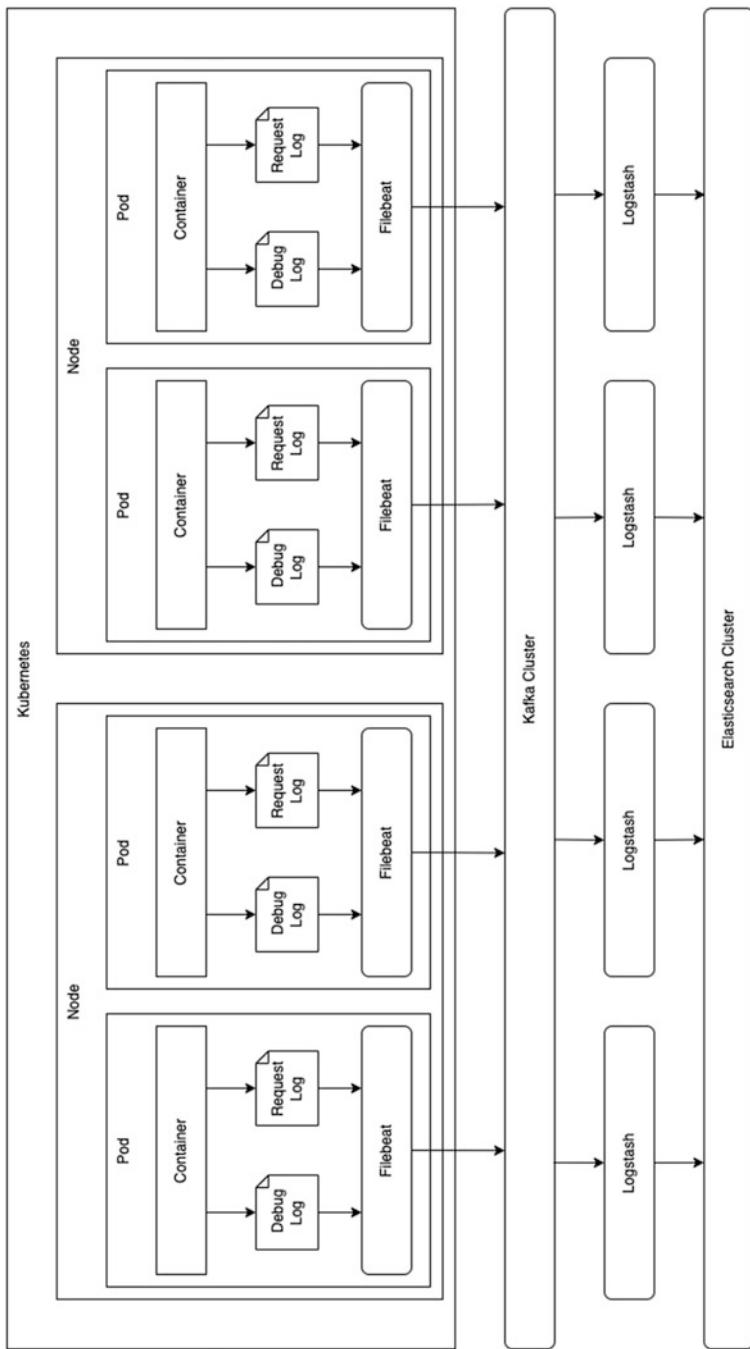


Fig. 7.5 Log collection with sidecar mode in Kubernetes

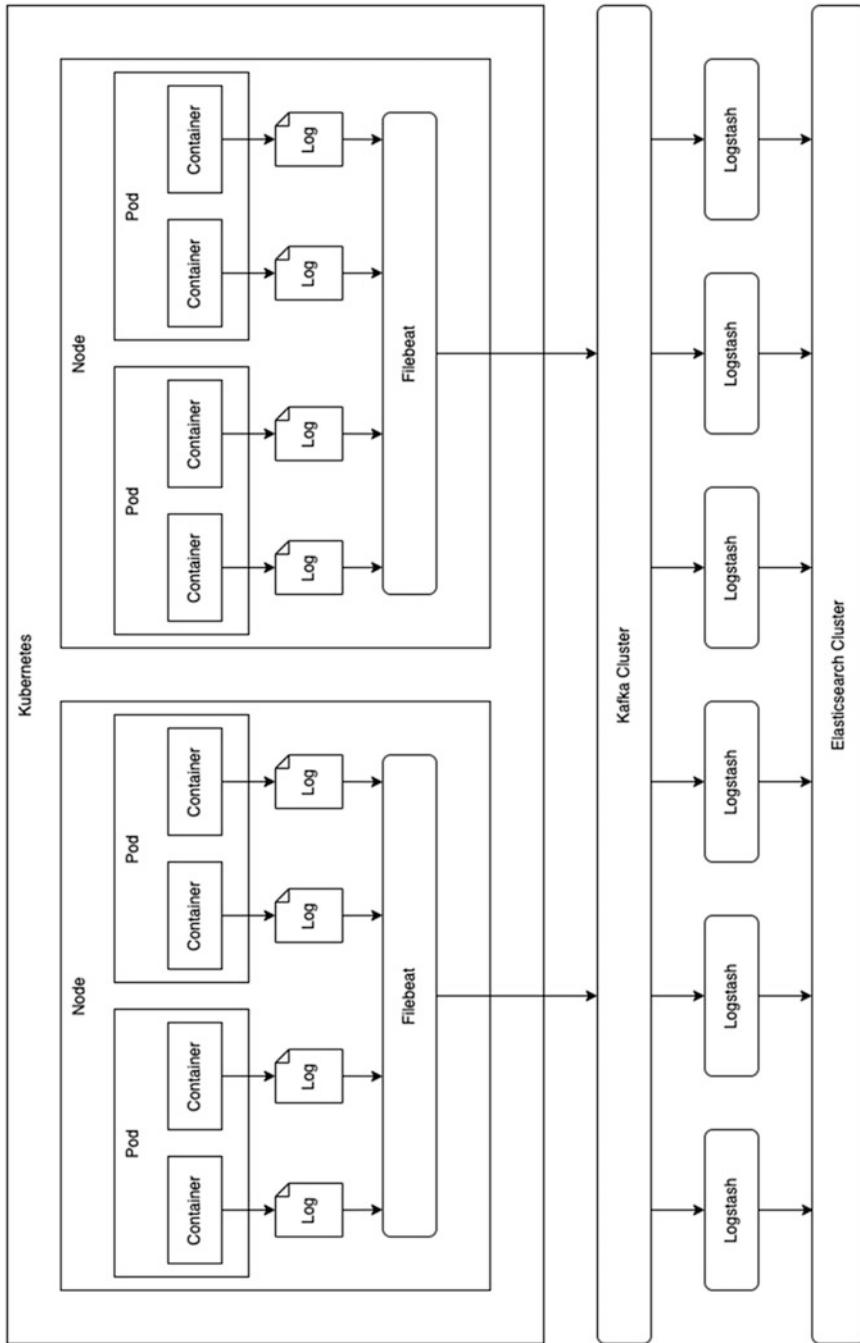


Fig. 7.6 Log collection with daemonset mode in Kubernetes

```
filter { # Cleanse log messages received from the Kafka cluster
if [message] =~ '^"/' { # Determine if the message field fetched from Kafka is in JSON format
json { # Extend internal fields and attributes to external
source => "message"
}
}
mutate { # Remove the original redundant message field
remove_field => ["message"]
}
}
if [log] =~ '/^"/ { # Determine if the log field fetched from the message field is in JSON format json
{ # Extend internal fields and attributes to external source => "
log"
}
}
mutate
{ # Remove the original redundant log field
remove_field => ["log"
]
}

# If the log is a Request Log, you can get the type field with the value request
if [type] == 'request' {
    json { # Parse the msg field in the request log as JSON, extending the internal fields and attributes to the external
source => 'msg'
}
date { # Use the Timestamp field recorded in the Request Log as the log timestamp
match => [ 'Timestamp', 'UNIX_MS' ]
}
if [ RequestBody ] {
# If the RequestBody field exists, recompress its JSON formatted content to a string
json_encode {
source => '[RequestBody]
'
}
} if
[ReplyBody] { # If the ReplyBody field exists, recompress its JSON formatted recompress the content to a string
json_encode {source
=> '[ReplyBody]
'
}
}
mutate {
remove_field => [ 'msg' ] # Remove the msg field that will not be used and logged again
}
} elsif [type] == 'debug' { # If it is a Debug Log, get the type field with the value debug
date { # Use the time field logged in the debug log as the log timestamp
match => [ 'time', 'ISO8601' ]
}
if [msg] =~ '/^"/ { # Determine if the msg field from the log field is in JSON format
json { # If it is in JSON format, extend the internal fields and attributes to the external
source => 'msg'
}
}
mutate { # The internal fields and attributes externally to remove the original redundant msg fields
remove_field => [ 'msg' ]
}
} } else {
date { # For other logs, such as third-party logs, try using the Timestamp field to get the timestamp
match => [ 'Timestamp', 'UNIX' ]
}
}

output { # Filtered log output configuration
if [type] == 'request' { # For request log output processing
elasticsearch {
hosts => [ "elasticsearch.host" ]
# Use the container name in the Kubernetes metadata to match the corresponding Request Log
index => "fw-%{[kubernetes][container][name]}-request-log-%{+YYYYww}"
manage_template => true
template_name => 'fw-ui-request-log' # Elasticsearch template name for Request Logs
}
```

```

# Elasticsearch template description file for Request Logs
template => '/logstash/template/fw-ui-request-log-template.json'
template_overwrite => true
} elseif [type] == 'debug' or [kubernetes][annotations][log-collection] { elasticsearch {
hosts => ["elasticsearch.host"]
    # Use the container name in the Kubernetes metadata to match the corresponding Debug Log
index => "ui-%{[kubernetes][container][name]}-debug-log-%{+YYYY.ww}"
manage_template => true
template_name => 'fw-ui-debug-log' # Elasticsearch template name for Debug Log
template => '/logstash/template/fw-ui-request-log-template.json'
    template_overwrite => true
}

```

Finally, Logstash sends the processed log messages with their corresponding Elasticsearch indexes to the Elasticsearch cluster for storage.

Thanks to the widespread use of Debug Logs and Request Logs and the unification of log formats and bodies, we can standardize the configuration of Logstash. At the same time, because of the above matching approach, new services can use ELK Stack completely transparently and write application-generated logs to Elasticsearch without any further configuration changes. This provides us with great convenience in building various microservices and reduces the maintenance and configuration costs of individual services.

7.2.3 *Displaying Logs with Kibana*

Kibana in the ELK Stack is a great tool for displaying Elasticsearch.

Based on Kibana, we cannot only search and view logs with its Discover feature but also analyze and display log data in various dimensions based on its Visualize and Dashboard features to clearly and visually display the behavior and status of the business system. Below, we will briefly describe the common operations for processing logs in Kibana.

1. Create index pattern

After collecting the logs into the Elasticsearch cluster in the way described in Sect. 7.2.2, we can retrieve the Elasticsearch indexes by creating the corresponding Kibana index pattern through Kibana's Management feature.

After creating the index pattern, you can see the input box to define the index pattern as shown in Fig. 7.7. In the input box, you can assign the logs according to the name of the Elasticsearch index they belong to, or use the wildcard “*” to fuzzy match the indexes that meet the conditions.

For example, if an Elasticsearch index is created in a Logstash configuration with the pattern fw-ui-%{[kubernetes][container][name]}-request-log-%{+YYYY.ww} and if the %{[kubernetes][container][name]} represents a container named order, then the logs generated in week 1 of 2021 will be collected in fw-ui-order-request-log-2021.01 (where 01 is the week number).

As shown in Fig. 7.8, enter “fw-ui-order-request-log-*” in the input box to automatically match all Request Logs of the container named order.

Click the “Next step” button and the screen shown in Fig. 7.9 will appear. We can select the field used as the basis for time categorization in the Time Filter field name drop-down menu.

The time filter field is not required and can be ignored if the logs in this index mode are not sensitive to the time they are logged. However, as a logging system, it is usually necessary to indicate when the log occurred or when the described event occurred by corresponding time information, so configuring the time filter field is very convenient for filtering and sorting logs by timestamp or intervals, which helps observe and analyze logs.

By default, Kibana generates a unique ID for the created index pattern itself, e.g., d9638820-7e00-11ea-a686-fb3be75e0cbf. This automatically generated ID, while can guarantee uniqueness, does not visually reflect what its corresponding log is, and is not friendly for index maintenance later. Therefore, it is possible to customize the ID of the indexing pattern in this advanced option, as long as the ID is unique. For ease of management, we recommend keeping this ID consistent with the name of the index pattern.

Continue to click the “Create index pattern” button in the lower right corner of Fig. 7.9 to complete the creation of the index pattern. In the upper right corner of the index pattern screen, there are three icon buttons, as shown in Fig. 7.10.

- (a) Star icon: Indicates that this index mode is the default focus index mode, and its content will be displayed by default in the Discover feature.
- (b) Refresh icon: Updates the list of fields for this indexing pattern. When a field changes in the log corresponding to the index pattern, clicking this icon button allows the field information displayed in Kibana to be updated, facilitating the operation of the changing field in other Kibana functions.
- (c) Delete Icon: Click this icon button to delete the current index mode.

In the main body of the page, you can see three different tabs, the Fields tab, the Scripted fields tab, and the Source filters tab.

The Fields tab shows the Name, Type, Format, Searchable, Aggregatable, and Excluded information for each field in the newly created index pattern. Clicking on the Pencil icon to the right of a field takes you to a page where you can edit the format and Popularity for that field. The Format drop-down menu provides the formatting options available for the field type so that the field is displayed in a more logical format in Kibana for easier reading; the Popularity value affects the order in which the field is displayed in Kibana; the higher the value, the more forward it is displayed, which facilitates quick orientation during the daily review. The Name, Type, Format, Searchable, and Aggregatable are determined by the corresponding Logstash configuration, Elasticsearch template configuration, and auto-detection settings and cannot be changed in Kibana. Exclusion is configured in the data source filter.

A Scripted field is a new field obtained by analyzing and processing an existing field. Scripted processing provides a quick and convenient way to display and aggregate, eliminating the cost of secondary editing and processing in other functions. However, it is important to note that scripted fields can only be used for

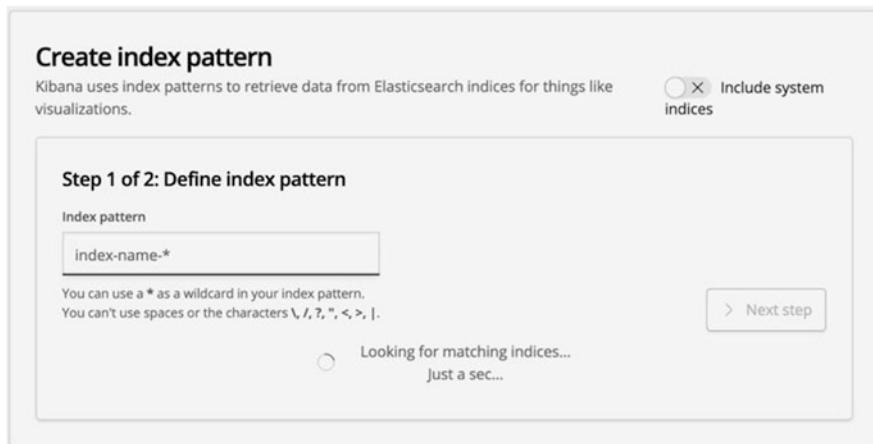


Fig. 7.7 Create index page

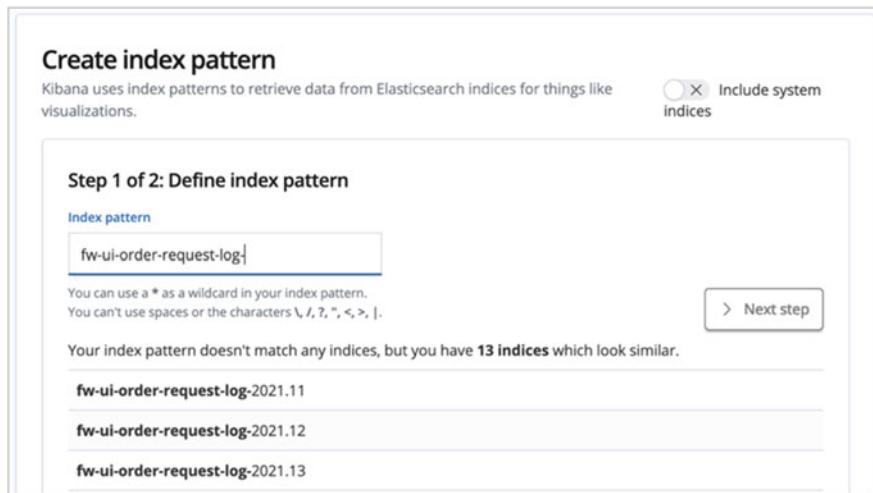


Fig. 7.8 Input index pattern

displaying log content, not for searching. Moreover, the fields added by scripting consume some computing resources, which may affect the performance of the related operations. If an error occurs in the editing script, all parts using this indexing mode will generate errors and make Kibana very unstable. Therefore, it is important to use this feature with sufficient care to understand the intent and risk of the operation at hand.

Figure 7.11 shows the Create scripted field interface. The scripted field supports the Painless scripting language and Lucene Expression language,

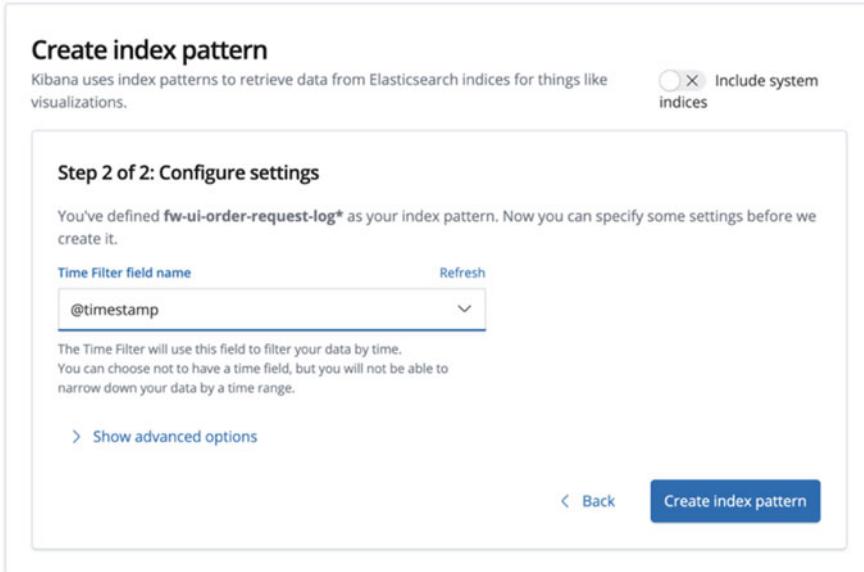


Fig. 7.9 Index configuration setting

which can be selected in the Language drop-down option. As for the specific scripting language, interested readers can check the official Elasticsearch website.

In the Source filters tab, you can set filter conditions by configuring keywords and wildcards to exclude fields contained in the index pattern from being displayed in the corresponding Discover and Dashboard features. After adding the filter conditions, jump back to the Fields tab to see the exclusion properties of the matched fields, as shown in Fig. 7.12, which shows the result of filtering using the “debug*” string.

2. Query log

Once the index pattern is created, we can use the Discover feature in Kibana to search.

As shown in Fig. 7.13, the index mode drop-down menu allows you to search and select the index mode corresponding to the log you need to query. After selecting the index mode, the system will automatically display the matching log content, and if the log content is configured with a time field for categorization, the throughput histogram of that log within the event interval will be displayed, and the bottom of the page will show the specific log content, and the fields contained in it.

If there is a need to repeat a query for a particular search, we can save the current search by clicking the “Save” button at the top of the screen in Fig. 7.13. The Save function can be used not only in the Discover function but also in the Visualize and Dashboard functions.

3. Log visualization

The screenshot shows the Kibana field list interface for the 'fw-ui-order-request-log-*' index. At the top, there are three tabs: 'Fields (57)', 'Scripted fields (0)', and 'Source filters (0)'. Below the tabs is a search bar labeled 'Filter' and a dropdown menu 'All field types'. The main area displays a table with the following data:

Name	Type	Format	Searchable	Aggregat...	Excluded
@timestamp	date		•	•	
@version	string				
Context.Deadline	number		•	•	

Fig. 7.10 Field list of index

Quickly querying logs via the Discover feature is just the basic functionality Kibana provides for log data stored in Elasticsearch, but Kibana is even more powerful because it has log visualization capabilities.

We can use the Visualize feature to count the throughput and response time of a certain service and then observe the working status and performance of the system. It is also possible to divide the logs by finer dimensions, such as analyzing the frequency of different customers' access to different methods, service effectiveness, and business function dependency by access method, customer ID, etc. What's more, we can discover the errors generated in the logs in real-time and categorize the errors according to the error codes, observe the problems that have occurred and are occurring in the system, and even analyze the usage behavior of customers through the scenarios generated by the error logs. In the following, we will briefly introduce the construction of visual charts, using the line chart in the Visualize feature as an example.

Select the Visualize module from the list on the left side of the Kibana interface, create a line chart on the page, and then select the index pattern that has been created or a saved search to use as a data source to create a visual chart. The line chart editing page is shown in Fig. 7.14. In the Data tab, the count of logs will be used as the Y-Axis indicator.

Continue to create the line chart as shown in Fig. 7.15. First, click the “X-Axis” inside the Buckets column to customize the horizontal coordinates, select “Date Histogram” in the Aggregation drop-down menu, select “@timestamp” in the Log field drop-down menu, select the time categorization field “@timestamp” to be used when configuring the indexing mode, select the default “Auto” in the Interval drop-down menu, and check the Drop partial buckets option to avoid statistical errors caused by incomplete first and last data.

Create scripted field

⚠ Proceed with caution
Please familiarize yourself with [script fields](#) and with [scripts in aggregations](#) before using scripted fields.

Scripted fields can be used to display and aggregate calculated values. As such, they can be very slow, and if done incorrectly, can cause Kibana to be unusable. There's no safety net here. If you make a typo, unexpected exceptions will be thrown all over the place!

Name

Language

Type

Fig. 7.11 Create scripted field

Expand the Y-axis list in Fig. 7.14, change the Aggregation drop-down menu from the default Count to Percentiles, select Duration in the Field drop-down menu, and, for the Percentiles list, we use both 95% and 99% values to indicate the response performance of the system, as shown in Fig. 7.16.

After editing, click the Play button (the right triangle icon button framed in Fig. 7.16), and the response line graph will be drawn on the right side of the screen. The Metrics & Axes tab and the Panel Settings tab provide several customizations related to the line graph presentation that we will not discuss in detail here. The edited line chart can be saved by clicking the “Save” button at the top of the screen.

4. Dashboard panel composition

Through the above process, we create visual charts depicting the state of the system, but a single chart usually does not provide a comprehensive picture of the system state, so Kibana provides dashboards that can hold multiple visual charts and quick searches at the same time.

Initialize an empty dashboard in Kibana. By adding a panel where we can find and add previously created visual charts and shortcut searches, the effect is shown in Fig. 7.17.

The added visual charts and shortcut searches can be resized and repositioned in the dashboard by dragging and dropping them, and the corresponding components can be quickly viewed, edited, and deleted by clicking the Settings button (a gear-shaped icon button) in the upper right corner of the panel. With this easy WYSIWYG operation, we can store the constructed dashboard for subsequent use. By unifying visual charts and quick searches into one dashboard, we can view the status of the system over different time frames.

Source filters

Source filters can be used to exclude one or more fields when fetching the document source. This happens when viewing a document in the Discover app, or with a table displaying results from a saved search in the Dashboard app. Each row is built using the source of a single document, and if you have documents with large or unimportant fields you may benefit from filtering those out at this lower level.

Note that multi-fields will incorrectly appear as matches in the table below. These filters only actually apply to fields in the original source document, so matching multi-fields are not actually being filtered.

Filter	Matches
debug*	No items found

Fig. 7.12 Source filter

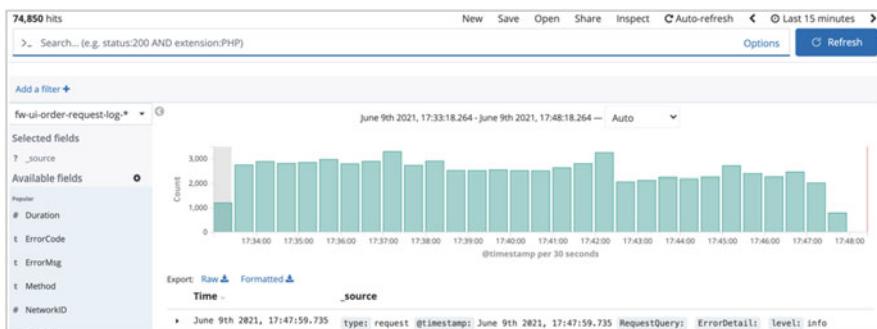


Fig. 7.13 Log display

5. Deep copy of the dashboard

Once we are familiar with the main features of Kibana, we can implement the daily observation and maintenance of the system state, but the above creation process is relatively tedious, and, for applications with dozens of microservices, the creation of dashboards becomes a burden to some extent. In applications where the Debug and Request Log formats are almost identical across services, it is easy to replicate new dashboards based on the existing ones.

In the Dashboard feature, Kibana provides a copy feature, but the copy action only creates the dashboard and the visual charts, and quick searches are not copied at the same time. This shallow copy is not sufficient for mapping existing dashboards to new service logs, and a deep copy of the dashboard requires another approach.

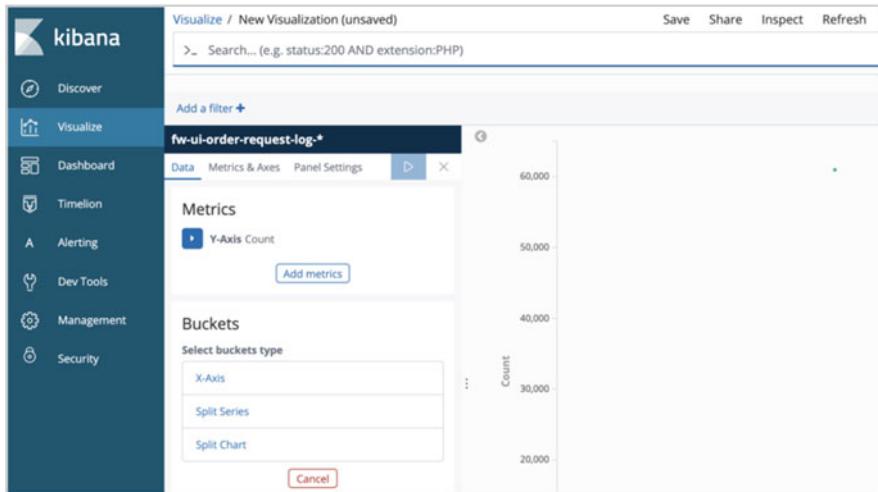


Fig. 7.14 Create new visualization

As shown in Fig. 7.18, a “Saved Objects” sub-function can be found in the list management function on the left side of Kibana, which allows us to query the index pattern, search, visualization, and dashboard that have been created.

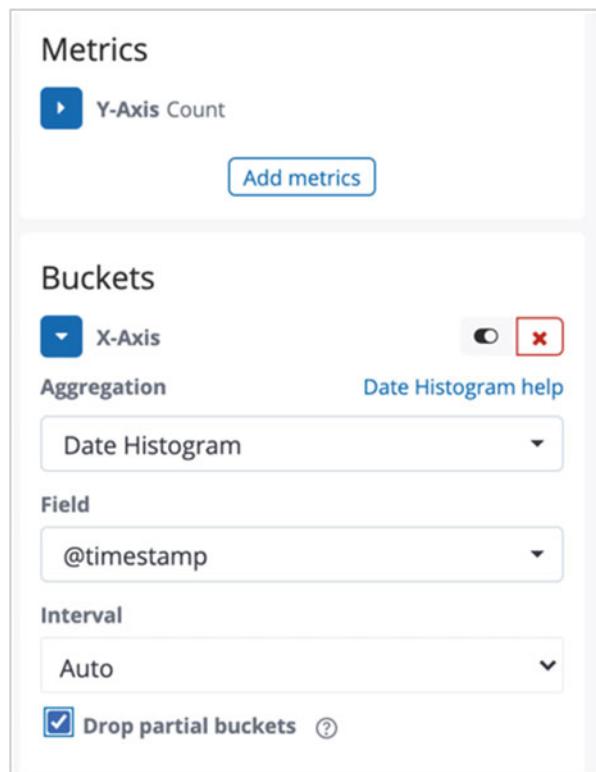
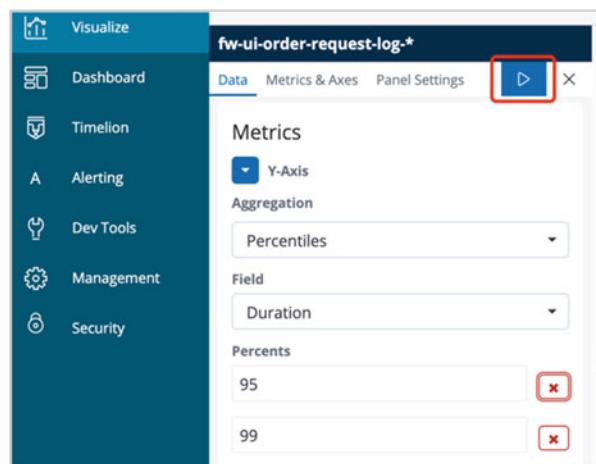
Clicking on different types of object links will display the corresponding text-based editing interface. In it, you can directly edit the properties and contents of the saved object and even modify the object ID.

After creating the index pattern, go to the corresponding modules of Discover, Visualize, and Dashboard to create new instances by Save As and shallow copy operations, respectively. The ID of the new instance can be obtained from two places; one is the corresponding display page and the other is the address bar of the saved object’s function screen. Then, we can modify the content layer by layer through the sub-functions of the saved object, in the order of Search, Visualization, and Dashboard, to achieve deep copy.

The above process is tedious but much faster than manually creating a dashboard from scratch, especially if you need to make deep copies of many existing objects.

Further, it can be seen that the Saved Objects sub-function of the Management function supports both exporting objects to and importing objects from files. By looking at the export file, you can see that the saved objects are described as JSON format files. Therefore, the IDs of the saved objects can be edited directly in the file.

Unlike in the edit page, we can also make Kibana automatically create new saved objects after receiving an imported JSON format file by deleting the `_id` field. Moreover, it is also possible to make a deep copy of the index pattern by exporting/importing it. In this way, the deep copy is still processed layer by layer in the order of index pattern, then search, then visualization, and finally

Fig. 7.15 Define metrics**Fig. 7.16** Input metrics data

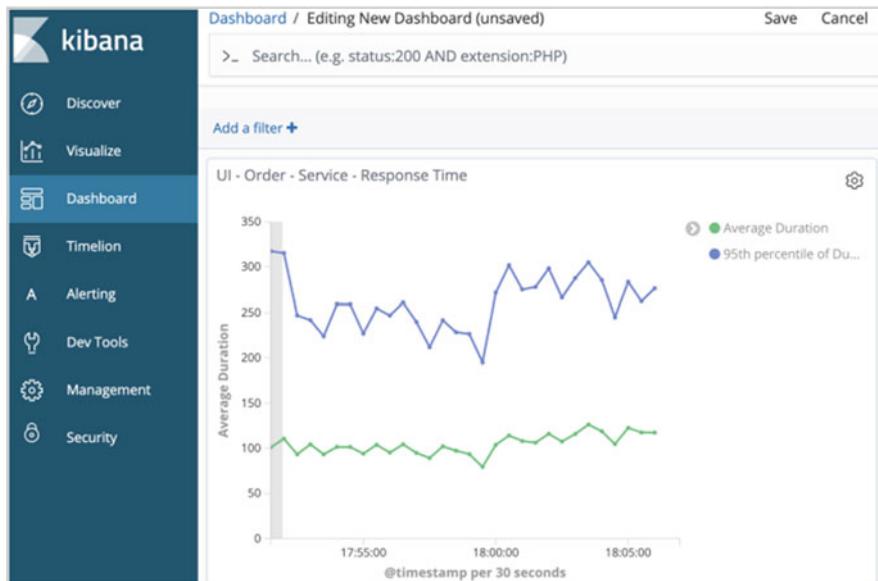


Fig. 7.17 Create new dashboard

dashboards. This is a much faster way than the previously described way of editing in Kibana. The corresponding JSON content is as follows:

The screenshot shows the Kibana interface with the sidebar navigation open. The 'Saved Objects' option is selected. The main area displays a table titled 'Saved Objects' with columns for 'Type' and 'Title'. A search bar at the top contains the query 'order service type:(dashboard)'. To the right of the search bar is a 'Type' dropdown menu with the following options:

- ✓ dashboard (66)
- index-pattern (43)
- visualization (326)
- search (101)

Fig. 7.18 Saved dashboard

... Through deep copy, with the guarantee that the basic visual diagram will not be lost, you can also create more personalized views of the characteristics of the new service, facilitating daily development and maintenance, and reducing the cost of learning, building, and maintenance due to the different forms of log formats.

7.3 Distributed Tracing

With the adoption of microservices in business systems, we can feel the benefits brought by microservices, and it is more convenient to maintain individual microservices after disassembling the system. However, due to the complexity of the business, the number of services in the system is increasing, and the deployment is becoming more and more complex, so we have to face problems such as complicated service invocation chains and difficulties in online tracing and debugging. To solve these problems, we built a full-process distributed tracing system.

7.3.1 *Core Concepts of Distributed Tracing System*

Distributed Tracing System can be used to trace and locate problems and analyze service invocation chains, etc. in microservices systems. The system data model was first introduced by Google in its paper “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure” published in April 2010, which contains the following components:

- Trace: A trace represents a complete invocation chain for a request in a distributed system—from its initiation to its completion, each trace will have a unique trace ID.
- Span: A small unit of work in a trace, either a microservice, a method call, or even a simple block of code invocation. A span can contain information such as start timestamps, logs, etc. Each span will have a unique span ID.
- Span Context: A data structure containing additional trace information. The span context can contain the trace ID, span ID, and any other trace information that needs to be passed to the downstream service.

How does the distributed tracing system locate the problem when invoking across services? For a client invocation, the distributed tracing system generates a trace ID at the request entry point and uses this trace ID to concatenate the invocation logs into each service to form a sequence diagram. As shown in Fig. 7.19, suppose the two ends of service A represent the beginning and end of a client invocation, and in between they pass through back-end services like B, C, D, E, etc. If there is a problem with service E, the problem will be quickly located without involving services A, B, C, and D to investigate and resolve the problem.

The most common scenario for distributed tracing is to quickly locate online problems as described above, with the following other typical application scenarios:

- Generate service invocation relationship topology diagrams to optimize the invocation chains.
- Analyze the performance bottlenecks of the entire application and optimize them in a targeted manner.

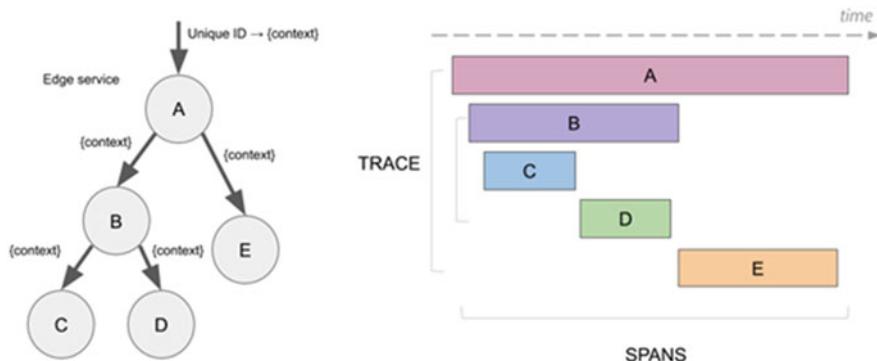


Fig. 7.19 Using tracing to locate problem in a distributed system

- Perform user behavior path analysis based on the complete invocation chain and then optimize the services.

7.3.2 Jaeger-Based Tracing Solution

There are already some mature solutions for distributed tracing system; we need to measure these existing solutions and select the right one for our business system.

1. Technology selection

Based on the current situation and requirements of the business system itself, we have identified several indicators in Table 7.1 as references for technology selection.

Based on the above indicators, we investigated the mainstream open-source projects on the market, including Jaeger, Zipkin, and so on. Also taking into account the market share, project maturity, project fit, and other factors, we finally chose Golang-based solution—Jaeger.

Jaeger tracing framework contains the modules shown in Table 7.2. The official Jaeger-client library provides support for most major programming languages and can integrate with business code, and Jaeger also supports analyzing the invocation relationship between services through Spark Dependency Job.

2. Practice on adoption and optimization

The first step to implementing a new system is usually to analyze the existing technical assets and reuse existing functions, modules, operation and maintenance environments, etc. as much as possible, which can greatly reduce the subsequent maintenance costs. The existing base environment of our business platform includes the following aspects.

Table 7.1 Indicator description

Indicators	Description
Does it support Golang client	Our business systems make extensive use of Golang as a development language, which makes us prefer the convenient Golang client solution
Whether it supports gRPC	Each business module mainly communicates with each other through the gRPC protocol, and whether it supports gRPC is an important reference indicator
How tracing data is stored	How to store, analyze, and present the trace data to developers clearly and concisely? These are also important indicators to consider in the selection process
Extensibility	As the infrastructure is upgraded and transformed, new components are constantly being introduced. Whether the solution supports infrastructure such as Istio and is easily extensible needs also to consider
Whether it is open-source	Cost and ease of customization and secondary development is also an important indicator, and we prefer open-source solutions

Table 7.2 Module description

Module	Description
Agent	As a daemon on the host, the agent is used to listen for trace packages from the server and then send them to the collector in bulk. The proxy is deployed to all hosts and also implements functions such as routing and load balancing for the collector so that all requests are not sent to the same collector entity
Collector	The collector is used to collect trace packages from agents and perform a series of processing on them, including trace package validation, indexing, format conversion, etc., and, finally, store them to the corresponding data warehouse. The collector can integrate with storage services such as Cassandra and Elasticsearch. In subsequent versions of Jaeger, Kafka support is added, and the injector Ingestor (a Kafka consumer) is provided to consume the data in Kafka
Query UI	The Query UI can provide a visual query service to retrieve and query the corresponding tracing ID from the data warehouse and visualize them
Spark dependency Jobs	The Spark dependency job reads raw data from Elasticsearch and streams the data in an offline manner to analyze the invocation relationship between services, the number of invocations, etc. The generated invocation relationship data can be displayed in the Query UI
Client Library	Used to integrate with business code

- (a) Diversification of protocols: need to support gRPC, HTTP, and other protocols.
- (b) Collect and analyze system logs through ELK + Kafka cluster.
- (c) The base runtime environment for microservices is based on Kubernetes + Istio. Except for a few special services that run on physical machines, most business services run in a Kubernetes cluster (AWS EKS), which means that each instance of a service runs as a Pod in the cluster.

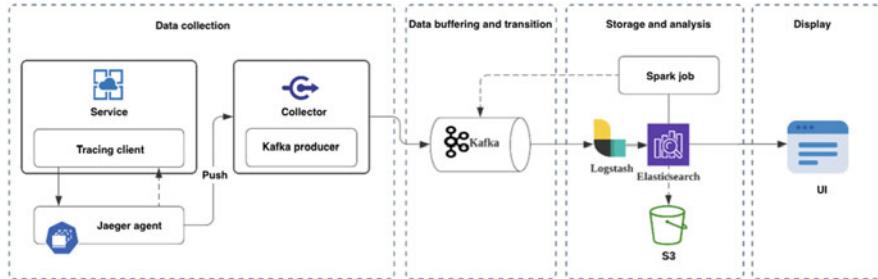


Fig. 7.20 Overall architecture of tracing system

Based on the above background, we designed the distributed tracing system implementation and modified some modules to integrate with the existing infrastructure.

First, the interfaces provided by each microservice to the outside world are not uniform; existing interfaces include gRPC, HTTP, and even WebSocket. Therefore, we did a wrapper layer on top of the official Jaeger client and implemented a custom tracing client that can hijack traffic for different communication protocols and inject tracing information into the request. We also added features such as filters (filtering traffic with given characteristics), trace ID generation, trace ID extraction, and compatibility with Zipkin Header. This part will be continuously updated as the platform continues to extend and upgrade.

In addition, to make full use of the company's existing Elasticsearch cluster, we used Elasticsearch as the back-end storage facility for the tracing system. Since the usage scenario is write-more and read-less, to protect Elasticsearch, we decided to use Kafka as a buffer. We extended the collector to process the data into Elasticsearch-readable JSON format, write to Kafka first, and then to Elasticsearch via Logstash.

In addition, for Spark dependency jobs, we have extended the output part to support importing data into Kafka.

Finally, due to the differences in the deployment environment within the microservice systems, we provide Kubernetes Sidecar, DaemonSet, and On-perm daemon process methods. The modified architecture is shown in Fig. 7.20.

Below is a detailed description of each part of the architecture diagram:

(a) Data collection

The data collection section mainly includes the tracing client, HTTP middleware, gRPC middleware, and request header information related to the Istio integration.

(i) Tracing Client

For Golang-based microservices, tracing information propagation within the service mainly relies on the span context. Our business systems generally support two communication protocols: HTTP and

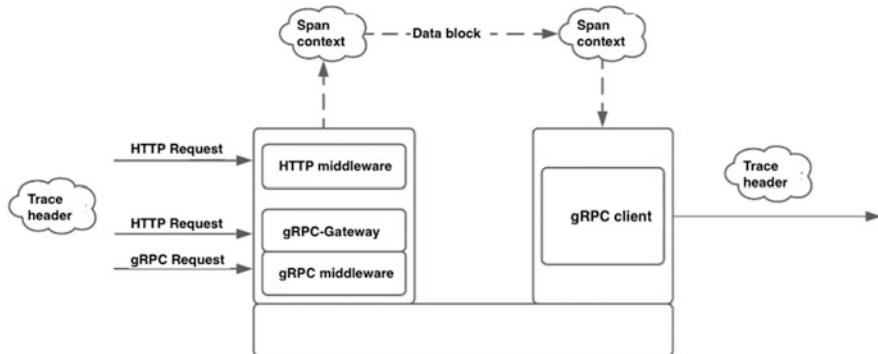


Fig. 7.21 Tracing information propagation process

gRPC. gRPC-Gateway automatically generates the HTTP interface. Of course, some services do not involve gRPC and expose the HTTP interface directly to clients. Here, HTTP is mainly to serve OpenAPI or front-end UI. gRPC is generally used to communicate between services. For this type of scenario, the tracing client provides the necessary components for the business microservice to use, and its propagation process is shown in Fig. 7.21.

For inbound traffic, the tracing client encapsulates HTTP middleware, gRPC middleware, and implements compatibility with the gRPC-Gateway component. For outbound traffic, the tracing client encapsulates the gRPC client. The “wrapper” here is not only a simple wrapper around the methods provided by the official Jaeger client but also includes support for features such as trace status monitoring, request filtering, etc. Requests such as /check_alive (service health probe) and /metrics (Prometheus metrics exposure) that do not need to be traced can be filtered out by the request filtering feature, and no tracing information is recorded.

(ii) HTTP middleware

The following code shows the basic definition and implementation of HTTP middleware:

```

func TraceHTTPMiddleware(tr opentracing.Tracer, h http.Handler, options ...nethttp.MWOption) http.Handler {
    if tr != nil {
        fn := func(w http.ResponseWriter, r *http.Request) {
            if !FilterOutRequest(r) {
                h.ServeHTTP(w, r)
            } else {
                traceHTTPMiddleware := nethttp.Middleware(tr, h,
                    options...)
                traceHTTPMiddleware.ServeHTTP(w, r)
            }
        }
        return http.HandlerFunc(fn)
    }
    return h
}

// FilterOutRequest filters out useless HTTP request URIs from the trace
// default filters out "/check_alive" and "/metrics"
func FilterOutRequest(req *http.Request) bool {
    for _, uri := range defaultFilterOutRequests {
        if req.RequestURI == uri {
            return false
        }
    }
    return true
}

```

(iii) gRPC Middleware

The following code shows the basic definition and implementation of the gRPC middleware:

```

func TraceGRPCClientMiddleware(opts ...grpc_opentracing.Option) grpc.UnaryClientInterceptor {
    if Enabled() {
        grpcOpts := []grpc_opentracing.Option{FilterOutGRPCMethodsOption(),
            grpc_opentracing.WithTracer(opentracing.GlobalTracer())}
        grpcOpts = append(grpcOpts, opts...)
        return grpc_opentracing.UnaryClientInterceptor(grpcOpts...)
    }
    return func(ctx context.Context, method string, req, reply interface{}, cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption)
        error {
            return invoker(ctx, method, req, reply, cc, opts...)
        }
}

// FilterOutGRPCMethodsOption filters out the specified method from the trace of the gRPC call
// filter out the "checkalive" health check related method
func FilterOutGRPCMethodsOption(methods ....string) grpc_opentracing.Option {
    return grpc_opentracing.WithFilterFunc(func(ctx context.Context, fullMethodName string) bool {
        for _, m := range defaultFilterOutGRPCMethods {
            if fullMethodName == m {
                return false
            }
        }
        for _, m := range methods {
            if fullMethodName == m {
                return false
            }
        }
        return true
    })
}

```

(iv) Istio Integration

Istio itself supports Jaeger tracing integration. For cross-service requests, Istio can hijack traffic of types such as gRPC, HTTP, etc. and generate the corresponding trace information. So, if you can integrate tracing information from business code with Istio, you can monitor the entire network information with complete tracing information inside the business and easily see how the request is processed in the network from the Istio Sidecar container.

to the business container. The problem here is that Istio integrates traces using the Zipkin B3 Header standard, which has the following format:

```
X-B3-TracelD: {TracelD}
{TracelD}X-B3-ParentSpanID: {ParentSpanID}
{ParentSpanID}X-B3-SpanID: {SpanID}
{SpanID}X-B3-Sampled: {SampleFlag}
```

The format of the FreeWheel Trace Header used within our business system is as follows:

```
FW-Trace-ID: {TracelD}:{SpanID}:{ParentSpanID}:{SampleFlag}
```

Since this Header is widely used in business code, integrating services such as logging and Change, it is difficult to be completely replaced at one time, so we rewrite the Injector and Extractor interfaces in the Jaeger client, which are defined as follows:

```
// The main role of the Injector interface is to insert the trace Header data into the context according to the pre-defined logic
type Injector interface {
    // Inject the SpanContext into the carrier
    Inject(ctx SpanContext, carrier interface{}) error
}

// The main role of the Extractor interface is to extract the trace information from the context
type Extractor interface {
    // Use the context as a carrier, extract the trace information from the Header and return a SpanContext object
    Extract(carrier interface{}) (SpanContext, error)
}
```

The newly implemented Injector and Extractor interfaces are compatible with both Zipkin B3 Header and FreeWheel Trace Header, and the service will prioritize checking the presence of the B3 Header when receiving a request and insert the FreeWheel Trace Header while generating a new span. FreeWheel Trace Header continues to be used within the service, and the B3 Header standard is primarily used for calls across services. Below is the request header information for the B3 Header standard:

```
X-B3-TracelD: {TracelD}
X-B3-ParentSpanID: {ParentSpanID}
X-B3-SpanID: {SpanID}
X-B3-Sampled: {SampleFlag}
FW-Trace-ID: {TracelD}:{SpanID}:{ParentSpanID}:{SampleFlag}
```

(b) Data buffering and transition section

As mentioned above, we choose Elasticsearch for data storage, and data collection and storage is a typical business scenario with more writing and less reading. For this kind of scenario, we introduce Kafka as the data buffer and transit layer. The official collector only supports writing data directly to Elasticsearch, so we modified the collector by adding a Kafka client component to convert span information into JSON format data and send it to Kafka, and then Logstash will process the data and store it in Elasticsearch. The following code shows the implementation process:

```
// Add Kafka dependencies
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>2.5.0</version>
</dependency>

// Convert span information into JSON format
private String toJson(List<Dependency> dependencyLinks) {
    ...
    ObjectMapper objectMapper = new ObjectMapper();
    json = objectMapper.writeValueAsString(new ElasticsearchDependencies(dependencyLinks, day));
    return json;
}

// Store the trace information to Kafka
private void storeToKafka(List<Dependency> dependencyLinks, String resource) {
    ...
    ProducerRecord<String, String> record = new ProducerRecord<String, String>(topic, "", toJson(dependencyLinks));
    producer.send(record);
}
```

The trace information stored in Elasticsearch has two main parts: the service/operation index and the span index.

The service/operation index is mainly used to provide a quick search for service and operation for the Query UI, and its structure is as follows:

```
// index structure
{
    "serviceName": "v3_adaptor",
    "operationName": "HTTP GET"
}
```

The span index is built based on the span, which is generated by the tracing client and contains the following main parts.

- (i) Basic trace information: such as traceID, spanID, parentID, operationName, duration, etc.
- (ii) Tags: mainly contains information related to business logic, such as request method, request URL, response code, etc.
- (iii) References: mainly used to indicate a span of the paternal-subordinate relationship
- (iv) Process: Basic information about the service
- (v) Logs: used to extend the business code

The following code shows the contents of the trace structure:

```
// Trace structure example
{
  "traceID": "5082be69746ed84a",
  "spanID": "5082be69746ed84a",
  "operationName": "HTTP GET",
  "startTime": ...,
  "duration": 616,
  "references": [
    {
      "refType": "CHILD_OF",
      "spanID": "14a9e000a96a2671",
      "traceID": "259f404f8409ad7"
    }
  ],
  "tags": [
    {
      "key": "http.url",
      "type": "string",
      "value": "/services/v3/**.xml"
    },
    {
      "key": "http.status_code",
      "type": "int64",
      "value": "500"
    },
    //...
  ],
  "logs": [],
  "process": {
    "serviceName": "your_service_name",
    "tags": [
      {
        "key": "hostname",
        "type": "string",
        "value": "xx-mac"
      },
      //...
    ]
  }
}
```

(c) Storage and analysis part

This layer is mainly used for the persistence and offline analysis of tracing data. When it comes to persistence, it is inevitable to consider the size of the data. Continuously writing a large amount of historical data to Elasticsearch will increase its burden, and the frequency of retrieval is relatively low for historical data that is too old. Here, we adopt an auto-archiving strategy to archive data more than 30 days to AWS S3. Elasticsearch only provides searching for relatively “hot” data.

The offline analysis is mainly used to analyze the span data in Elasticsearch. The structure of a span data contains its own tracing ID and the tracing ID of its parent node, and each node’s information will indicate which service it belongs to. Here, we only care about the call relationship between the spanning services, as shown in Fig. 7.22. Only nodes A, B, C, and E are considered for offline analysis, and node D is ignored because it only has a call relationship with node C, which is also in service 3.

(d) Display section

The display layer queries data from Elasticsearch, aggregates data with the same tracing ID, and renders it on the front end. From the presentation layer, you can clearly see how many different services a request went through (marked as different colors), the time taken by the request in each service, and the total response time of the request, as shown in Fig. 7.23.

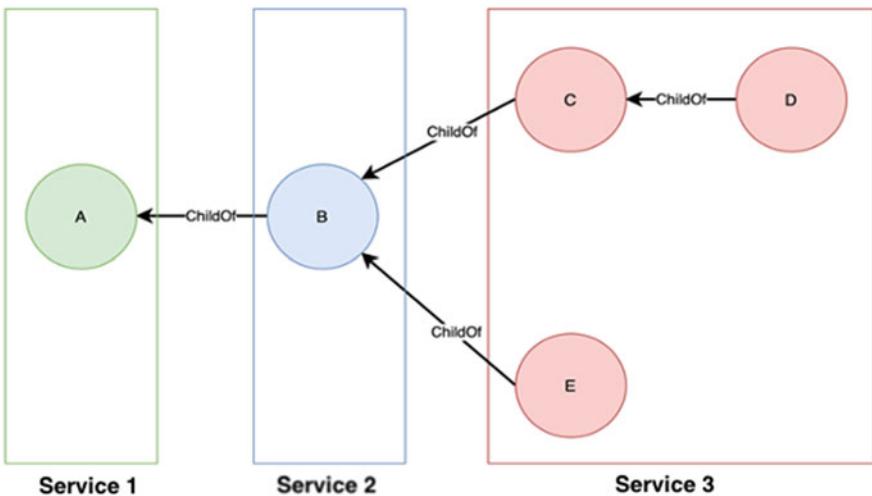


Fig. 7.22 Offline analysis



Fig. 7.23 Tracing display

7.4 Metrics

When we build a monitoring system, collecting and observing business logs can help us understand the current business behavior and processing status of the system in detail. However, a monitoring system that achieves complete observability also needs to collect, observe, and analyze additional metrics to achieve real-time control of operational status. These metrics include CPU usage, memory occupation, disk read/write performance, network throughput, etc. To enable these metrics to be collected and observed effectively, this section outlines how to collect metrics from the Kubernetes cluster and its internal running applications via Prometheus and visualize the collected metrics via Grafana.

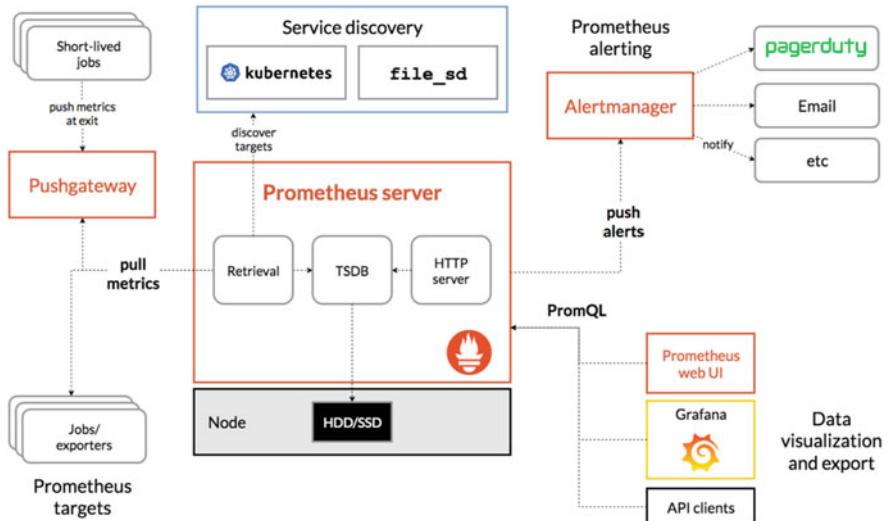


Fig. 7.24

7.4.1 Collecting Metrics with Prometheus

Prometheus is a very powerful suite of open-source system monitoring and alerting tools that have been used by a large number of companies and organizations with an extremely active community since it was built at SoundCloud in 2012. The project joined CNCF for hosted incubation in 2016 and successfully graduated in 2018. In this section, we will briefly describe the functional structure of Prometheus and how to use it to collect metrics within Kubernetes.

1. Introduction to Prometheus

Prometheus provides users with rich and convenient functions. It can define multidimensional data models by collecting the name and key-value pairs of time-series metrics and provides PromQL for flexible querying of multidimensional data. Each of its independent server nodes is highly autonomous and does not rely on distributed storage, and the configuration for the collection target is flexible, either through static files to manage the configuration or through a service discovery mechanism to get the configuration automatically. In addition, the robust ecosystem built by Prometheus includes a variety of core and extension components, most of which are written in Golang and can be easily compiled into static binaries and deployed.

The ecosystem architecture of Prometheus is shown in Fig. 7.24.

Prometheus enables direct HTTP-based pulling of metrics data through a detection plug-in running in the target application and indirect pulling of metrics data from Pushgateway, which is pushed by short-time tasks. It stores all the collected metrics data in an internal timing database, then aggregates and records

new time series with relevant processing rules, and generates the corresponding alerts for pushing to Alertmanager. The collected metrics can also be displayed and exported through Prometheus UI, Grafana, or API client.

Prometheus can easily record any purely digital time series for both physical-machine-centric and service-oriented architecture (SOA) application monitoring. In the microservices domain, its ability to collect and query multidimensional data makes it a perfect choice for metrics collection.

Designed with reliability at its core, Prometheus helps users quickly diagnose what's wrong when a system fails. Each Prometheus server is available on its own and does not depend on any network storage or remote services. When other components in the infrastructure are not working properly, it can still function without adding any extension components. However, because of the high emphasis on reliability at the same time, Prometheus will always ensure that statistical information is accessible, even in the case of incorrect data source information. If the system being built expects the observed metrics data to be 100% accurate, Prometheus will not be an optimal choice because it cannot guarantee the exhaustiveness and completeness of the collected metrics. For example, when building a billing system, the system will strictly require the accuracy of the data to ensure the correctness of the final billing, and then other systems will need to be used to count and analyze the relevant data.

Based on the design concept and functional features of Prometheus, we recommend it as an excellent monitoring tool for collecting business system metrics to measure and observe system operation.

2. Collect metrics for applications in a Kubernetes cluster

Prometheus provides client-side toolkits for most applications built via popular languages, and we can expose metrics data in our applications by serving HTTP interfaces. The following code shows the Golang library and the exposed metrics data:

```
import (
    "net/http"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)
func ServeMetrics() {
    // Register the Prometheus handler to the HTTP "/metrics" entry
    http.Handle("/metrics", promhttp.Handler())
    // Open and listen to HTTP requests on port 1234
    http.ListenAndServe(":1234", nil)
}
```

With the above simple configuration, Prometheus can provide the default metrics data for the application by visiting `http://localhost:1234/metrics`.

If the application is built with gRPC type, it is convenient to use the Prometheus interceptor provided in the gRPC-ecosystem libraries to count the metrics data associated with the gRPC service in the following example.

```

func InitGPRCServer() {
    ...
    // gRPC server initialization
    grpcServer := grpc.NewServer()
        // Fill in the Prometheus unary interceptor on the gRPC server side
        grpc.UnaryInterceptor(grpc_prometheus.UnaryServerInterceptor),
        // Fill in the Prometheus stream interceptor on the gRPC server side
        grpc.StreamInterceptor(grpc_prometheus.StreamServerInterceptor)
    )
    // Register the gRPC service implementation to the gRPC server side
    grpcService. RegisterGRPCServiceServer(s.server, &grpcServiceImpl{})
    // Register the gRPC server to Prometheus
    grpc_prometheus.Register(grpcServer)
    // Register the Prometheus handler to the HTTP "/metrics" entry
    http.Handle("/metrics", promhttp.Handler())
    ...
}

```

The Prometheus interceptor provided by the gRPC-ecosystem not only provides methods for the server side to obtain relevant service metrics but also provides an interceptor for use with gRPC clients. The following code shows how the interceptor is used:

```

func GRPCClientCall() error {
    ...
    clientConn, err := grpc.Dial(
        // gRPC server address

    grpc.WithUnaryInterceptor(grpc_prometheus.UnaryClientInterceptor),
        // Fill in the Prometheus monadic interceptor in the gRPC client

    grpc.WithStreamInterceptor(grpc_prometheus.StreamClientInterceptor),
    )
    if err != nil {
        return err
    }
    // Create gRPC client from connection information
    client := grpcProto.NewGRPCServiceClient(clientConn)
    // Send a request to the gRPC server via the gRPC client and get the
    return result
    resp, err := client.GetGRPCEntity(s.ctx, &grpcService.Request{ID: 1})
    if err != nil {
        return err
    }
    ...
}

```

In addition to this, we can further customize the way we register additional metrics, and four metric data types are available in the Prometheus client toolkit.

- (a) **Counter:** Used to record the accumulation of a value over time, which can only increase in one direction and not decrease, such as the number of visits to a service portal, the total size of processed data, etc.
- (b) **Gauge:** Used to record the instantaneous condition of a certain value; unlike a counter, it can either increase or decrease. We can use this type of data to record the current system load, concurrency, etc.

- (c) Histogram: Used to count and record the distribution of some index data; it can be used to observe the response performance of the system at a certain time or the distribution of return value types.
- (d) Summary: Very similar to the histogram, it is also used to record the distribution of metric data, but it uses percentages as a statistical criterion. It can be used to show the response of 50, 90, and 99% of the system's requests in a given period.

Here, we also take the Golang application as an example and add four types of custom metrics data to the ServeMetrics function method above, with the following code:

```

var (
    // Counter to record the number of function accesses
    funcCounter = promauto.NewCounter(prometheus.CounterOpts{
        // The name of the custom indicator, as a unique ID, cannot be
        // repeatedly registered
        Name: "func_test_for_prometheus_counter",
        // Description of this indicator
        Help: "Func test for prometheus counter",
    })
    // Record the number of functions being accessed
    funcGauge = promauto.NewGauge(
        prometheus.GaugeOpts{
            Name: "func_test_for_prometheus_gauge",
            Help: "Func test for prometheus gauge",
        })
    // Plot the function response duration histogram
    funcHistogram = promauto.NewHistogram(prometheus.HistogramOpts{
        Name: "func_test_for_prometheus_histogram",
        Help: "Func test for rometheus histogram",
        // Linear histogram partitioning, creating one statistical interval
        // at 200 intervals starting from 100, for a total of 5 statistical intervals
        Buckets: prometheus.LinearBuckets(100, 200, 5),
    })
    // Plot function response length summary
    funcSummary = promauto.NewSummary(prometheus.SummaryOpts{
        Name: "func_test_for_prometheus_summary",
        Help: "Func test for prometheus summary",
        // Quantile settings for statistics, where keys indicate quantile
        // criteria and values indicate error ranges
        Objectives: map[float64]float64{0.5: 0.05, 0.9: 0.01, 0.99: 0.001},
    })
)

func FuncTestForPrometheus() {
    funcCounter.Inc() // add one to the access counter when the function
    is accessed
    funcGauge.Inc() // add one to the meter when the function is accessed
    start := time.Now() // record the timestamp when the function is
    accessed
    ...
    duration := float64(time.Since(start).Milliseconds())
    funcHistogram.Observe(duration)
    funcSummary.Observe(duration)
    funcGauge.Dec()
}

```

When we create multiple Golang concurrent (Goroutine) calls to the above functions, we can obtain the indicator data as shown below by accessing the indicator data portal at <http://localhost:1234/metrics>:

```
# HELP func_test_for_prometheus_counter Func test for prometheus counter#
TYPE func_test_for_prometheus_counter
counterfunc_test_for_
prometheus_counter 1972.0 # The counter recorded 1972 visits
# HELP func_test_for_prometheus_gauge Func test for prometheus gauge # TYPE
func_test_for_
prometheus_gauge
gaugefunc_test_for_prometheus_gauge
3.0 # There are 3 functions being called for the gauge statistics #
HELP func_test_for_prometheus_histogram Func test for
prometheus
histogram# TYPE func_test_for_prometheus_histogram
histogramfunc_test_for_prometheus_histogram_bucket{le="100.0"} 438.0
# Histogram statistics response time is less than 100ms 438 times
....
```

Once we can get the metrics data through the HTTP portal, we can configure the metrics collection related configuration through the `prometheus.yml` file in Prometheus. If you want to display the specified Prometheus collection object, you can configure it as follows:

```
scrape_configs:
- job_name: test_for_prometheus # name of the collection task
scrape_interval: 10s # collection interval, if undeclared, global configuration
scrape_timeout: 60s # collection
timeout threshold, if undeclared, global configuration
scheme: http # Protocol type, either HTTP or HTTPS, if undeclared, HTTP
metrics_path: /metrics # Collection portal, if undeclared, default portal/metrics
static_configs: # Static configuration items
- targets: # Target addresses
-
localhost:1234 # domain name and port number of the HTTP address
```

If you want Prometheus to use the service discovery mechanism to automatically collect the metrics data entry exposed inside the Kubernetes cluster, you can configure the `prometheus.yml` file as follows:

```

scrape_configs:
- job_name: test_for_prometheus # name of the collection task
scrape_interval: 10s # collection interval, if undeclared, global configuration
scrape_timeout: 60s # collection
timeout threshold, if undeclared, global configuration
scheme: http # Protocol type, either HTTP or HTTPS, or HTTP if not declared
kubernetes_sd_configs: # Kubernetes service discovery configuration
- role: pod # Collect application metrics data exposed in Pods
relabel_configs:
  # Collect metrics only for services that have prometheus.io/scrape configured in service
  annotations with a value of true
  - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape] action: keep
    regex: true
    # Use the
    prometheus.io/path value configured in the service
    annotation as the metric_path
  - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
    action: replace
    target_label: __metrics_path__ regex
    :(.+
    +)
  - source_labels: [__address__] # Extract service address information stored in address action
    :keep
    regex: ([^:]+);(\d+)
    - source_labels: __address__ __meta_kubernetes_pod_annotation_prometheus_io_port
      # Use the prometheus.io/port value configured in the service annotation as the port
    action: replace
    regex: ([^:]+)(?::\d+)?;(\d+)
    replacement: $1:$2
    target_label: __address__ # Combine the old server address and the new port into a new address
  - source_labels: [__meta_kubernetes_namespace] # Record the Kubenete namespace
    action: replace
    target_label
    :namespace - source_labels:
    __meta_kubernetes_pod_name] # Record the collected Pod names
    action: replace
    target_label: pod_name
    # In the Kuberneets Helm configuration of the service, we need to configure the corresponding annotations
spec:
template:
metadata:
annotations:
  prometheus.io/scrape: true # Collect metrics data generated by the current application
  prometheus.io/path: /metrics # Metrics data collection portal
  prometheus.io/port: 1234 # Port for metrics data collection

```

3. Collect metrics for tasks in the Kubernetes cluster

Unlike regular service-based applications, task-based applications cannot exist and maintain the corresponding metrics data interface for a long time. Therefore, we need to make use of the Pushgateway in Prometheus to push the metrics data to be recorded from the task to the gateway, and then the Prometheus server will pull and collect them. Below, we also take Golang application as an example to briefly introduce how to implement metrics collection through the Pushgateway in Prometheus toolkit with the following code.

```

var (
    // Record function runtime, metrics created via
Prometheus package are not registered
duration = prometheus.NewGauge(prometheus.GaugeOpts{
Name: "func_test_for_push_gateway_duration"
,Help :"Func test for Pushgateway duration",
})
)

func FuncTestForPushGateway() error {
start := time.Now() // Record the timestamp of when the function was accessed
// ....
// Calculate the response time of the function and record it in the duration indicator
duration.Set(float64(time.Since(start).Milliseconds()))
// Create the pushed by the address localhost:9091 of the Pushgateway and the name of the current push task
if err := push.New("http://localhost:9091", "test_for_push_gateway").
Collector(duration). // Set the metrics to collect
Grouping("jobs", "test"). // Set the classification label
Push(); err != nil { // Push and process the returned error message
return err
}
}

```

The above methods push the metrics data of the current task to the corresponding gateway. In addition to the Push method used in the example, the toolkit also provides the Add method, which uses the HTTP PUT operation to replace all metrics with the same task name in the Pushgateway when submitting, and the Add method uses the HTTP POST operation to replace only those metrics with the same task name and the same grouping label in the Pushgateway when submitting. Therefore, if we need to submit indicator data more than once in the program logic, it is recommended to use the Add method to avoid overwriting errors caused by the submission.

Accordingly, for the Prometheus configuration, we also need to add a new collection task to collect the metrics data from the Pushgateway, with the following code:

```

scrape_configs:
- job_name: pushgateway # Job name of the metrics collected in
Pushgateway
static_configs: # Static configuration items
- targets: # Target address
- localhost:9091 # Domain name and port number of
Pushgateway
honor_labels: true # Retain information about the labels in the collected metrics

```

4. Collect Kubernetes cluster resource status metrics via kube-state-metrics

In addition to the metrics data of the applications running in Kubernetes, we also care about the operational state of the entire Kubernetes cluster, such as the health of the Pods, the number of services, the number of restarts, and so on. This is where we can use kube-state-metrics to listen. kube-state-metrics generates metrics data by fetching information from the Kubernetes API service and exposing the data in a format that meets the Prometheus specification in the HTTP service's "/metrics" portal of the HTTP service, enabling the collection of internal Kubernetes metrics data.

Because kube-state-metrics provides very friendly support for Prometheus, it is very easy to deploy in Kubernetes and collect metrics data automatically through the service discovery mechanism in the Prometheus configuration. The specific code is as follows:

```
# Pull kube-state-metrics repository information helm
repo add kube-state-metrics
https://kubernetes.github.io/kube-state-metrics/helm repo update

# Install the default Chart file for kube-state-metrics helm
install kube-state-metrics kube-state-metrics/kube-state-metrics --version 2.0.0
```

Once started, kube-state-metrics will maintain a /metrics portal via HTTP on port 8080 of the Pod it is running on to provide metrics data. The data can be collected by performing the following service discovery configuration in the Prometheus configuration:

```
scrape_configs:
- job_name: "kubernetes-service-endpoints" # Kubernetes service endpoint metrics sampling configuration
  kubernetes_sd_configs:
    - role: endpoints # Collection of exposed application metrics in endpoints
      relabel_configs:
        - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scraped]
          action: keep
          regex: true
        - action: labelmap # Match and keep service labels
          regex: __meta_kubernetes_service_label_(.+)
        -
        source_labels:
        [
          __meta_kubernetes_namespace
        ] # collected Kubernetes namespaces action
        : replace
        target_label: kubernetes_namespace
        - source_labels: [__meta_kubernetes_service_name] # Record the name of the collected service action
        : replace target_label
        label: service_name
```

7.4.2 Displaying Metrics with Grafana

Once the metrics have been collected through Prometheus, there are multiple ways to access and visualize the time-series data stored there. Although Prometheus provides its visualization interface, the way it presents content is very basic and not enough to meet daily monitoring needs. Grafana, an open-source project that also joins CNCF with Prometheus, has richer graphical presentation capabilities. It can receive data from various data sources such as Prometheus, Elasticsearch, MySQL, CloudWatch, Graphite, etc. and provides a large number of sample dashboards for different scenarios.

We can configure the Data Sources in the Configuration module provided by Grafana. Grafana will create an empty dashboard interface where we can add and edit the panels we need. As shown in Fig. 7.25, the panel editing page can be accessed after adding a new panel.

Let's take the example of building a response curve graph and briefly explain how to edit the panel. In the Query tab, select the data source as Prometheus via the drop-down menu and use PromQL in the indicator input box below to describe the search operation to be performed.

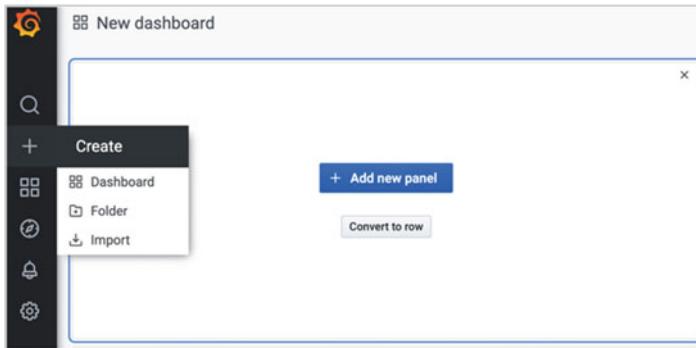


Fig. 7.25

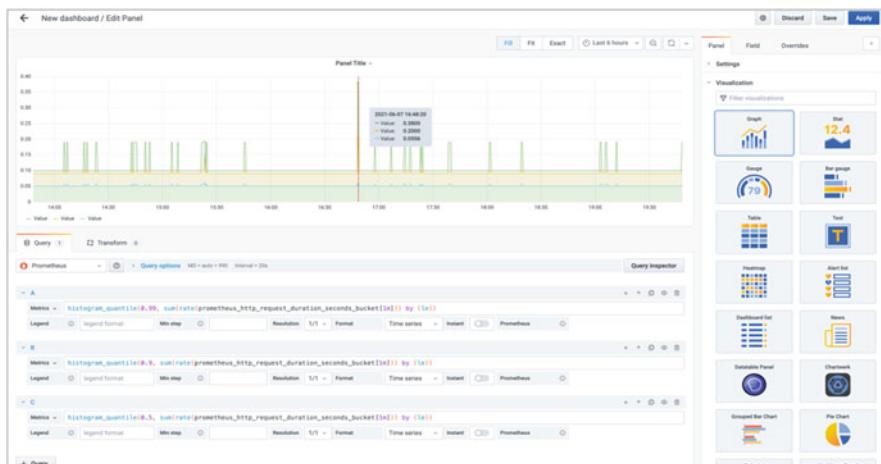


Fig. 7.26

Type $\text{histogram_quantile}(0.99, \text{sum}(\text{rate}(\text{prometheus_http_request_duration_seconds_bucket}[1m]))) \text{ by } (\text{le})$

($\text{prometheus_http_request_duration_seconds_bucket}[1\text{m}]$)) by (le)), which uses the $\text{histogram_quantile}(\varphi \text{ float}, b \text{ instant-vector})$ built-in function to calculate the φ quantile sample maximum of data b . Also, the rate ($\text{prometheus_http_request_duration_seconds_bucket}[1\text{m}]$) in the content of composition b indicates that the metrics data of HTTP request duration of Prometheus are sampled in a time window of 1 min, and then all metrics carrying le tags are aggregated by $\text{sum}(x)$ by (le). The le tag marks the upper limit of the sample split, through which we can filter out other irrelevant samples. With the above settings, the panel plots the 99 quintiles of Prometheus request responses. By further copying and modifying the parameters, we can quickly obtain its quintile and ninth quantile plots and aggregate them in the same panel, as shown in Fig. 7.26.

In the upper right corner of the dashboard interface, you can see the gear icon, which is the settings button that can be clicked to enter the dashboard information settings interface. The bottom of its navigation menu contains a JSON Model module, which enables the dashboard to be described in JSON format. We can make quick edits based on this, or we can quickly copy panels from other dashboards into the current dashboard.

Grafana also provides an Import option in the Create module of the main interface, which allows us to quickly import and copy existing dashboards directly, as well as a large number of dashboard samples for different systems and application scenarios, which can be imported directly when building dashboards.

7.5 Monitoring and Alerting

The previous sections of this chapter introduced the adoption of logs, distributed tracing, and metrics in the system, respectively. We chose different products to collect three pillars' data, and the three kinds of data stored in their respective products are not interoperable, which poses a challenge for establishing a unified monitoring and alerting system. This section will introduce how to design and implement an efficient monitoring and alerting system based on different products.

7.5.1 *Monitoring Platform*

Before introducing the setup of our business system monitoring platform, it is necessary to look at a few key prerequisites.

- Different products are used for logging, tracing, and metrics.
- The system is mainly focused on B2B business, with more than 30 existing microservices under the responsibility of different teams, deployed on AWS's EKS.
- From the company's organizational structure, there is a team of site reliability engineers, a monitoring platform infrastructure support team, an application system infrastructure support team, and a core business team.

Based on these conditions, we divide the overall monitoring platform into three layers from the bottom up, as follows:

- Infrastructure Monitoring: The operational status of these facilities, such as network, gateway, and EKS cluster (including node status, Istio, etc.), is monitored by the site reliability engineers and the underlying application system infrastructure support team, and this section will not cover this part of the content.
- Microservices generic monitoring mainly monitors the business system microservices generic indicators, including microservice Pod status (CPU,

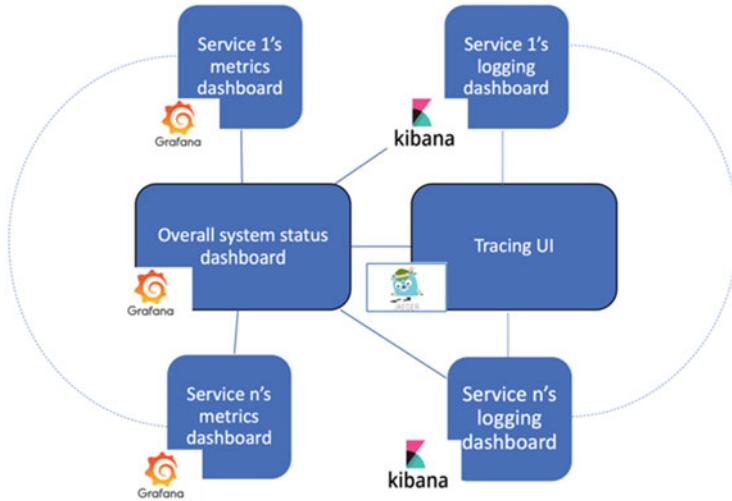


Fig. 7.27 Overall structure of monitoring system

memory usage, etc.), the number of requests processed, request latency and request results, etc.

- Business monitoring: Monitoring is defined according to each service's own business requirements, such as monitoring certain large customer-specific services.

1. Microservice Generic Monitoring

Microservices Generic Monitoring is the most important layer of monitoring in the business system, which can help us view the overall operation status of the business system, discover problems in time, quickly find the causes of problems, and solve them. Since the system uses three products, how to link them organically becomes the key to efficient monitoring. The overall structure of the final monitoring system is shown in Fig. 7.27.

First, each service creates a standardized dashboard to show its status, including a metrics dashboard (Grafana Dashboard) and a logging dashboard (Kibana Dashboard), while the logging data is associated with the tracing system. Finally, an overall system status dashboard is created to show the status of all microservices in the system, and this dashboard can be easily linked to individual service dashboards and tracing systems. In the following, we will describe how to build the microservice standardized dashboard and the overall system status dashboard.

(a) Microservice standardized dashboard

The standardized dashboard mainly consists of the following four golden signals (from Google's Site Reliability Engineer's Handbook).



Fig. 7.28 Standardized metrics dashboard

- (i) Traffic: The number of service requests, such as the total number of requests completed in a certain period, the number of requests per second (QPS), etc.
- (ii) Latency: The time taken to complete the request, including the average time, 95th percentile (p95), etc.
- (iii) Errors: The error request that occurred.
- (iv) Saturation: The saturation of the current service, which mainly checks the restricted resources, such as memory, CPU, and disk status. Usually, the saturation of these resources will cause a significant downgrade of the service performance or even stop the service.

Next, we will introduce how to build microservice standardized dashboards on Grafana and Kibana, which of course are based on the standardized collection of logs and metrics described in the previous sections of this chapter.

Build microservices standardized metrics dashboard on Grafana

As shown in Fig. 7.28, the whole dashboard is divided into two parts. At the top are several key metrics that can indicate whether the microservice is running properly, including the number and status of running Pods, the health of CPU and memory usage, and whether the request error ratio is normal. Below are the detailed data of the four golden metrics.

Here, we use Grafana's dashboard template to show the status of all services. First, we create a dashboard and add two custom variables through the dashboard: `k8s_cluster` and `service_name`, whose options are the cluster name of the business system hosted and the names of all

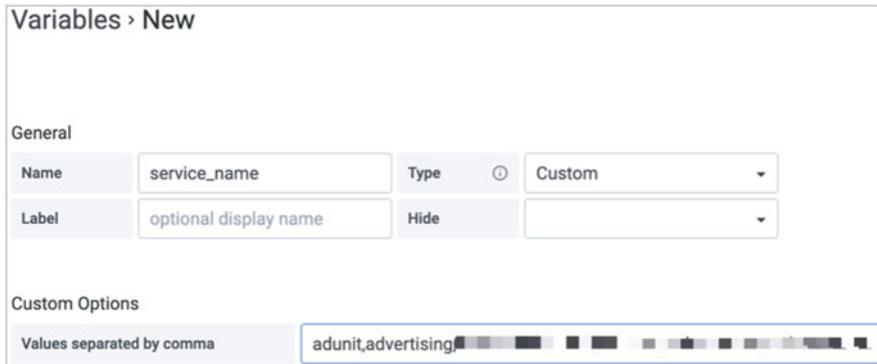


Fig. 7.29 Adding custom variable

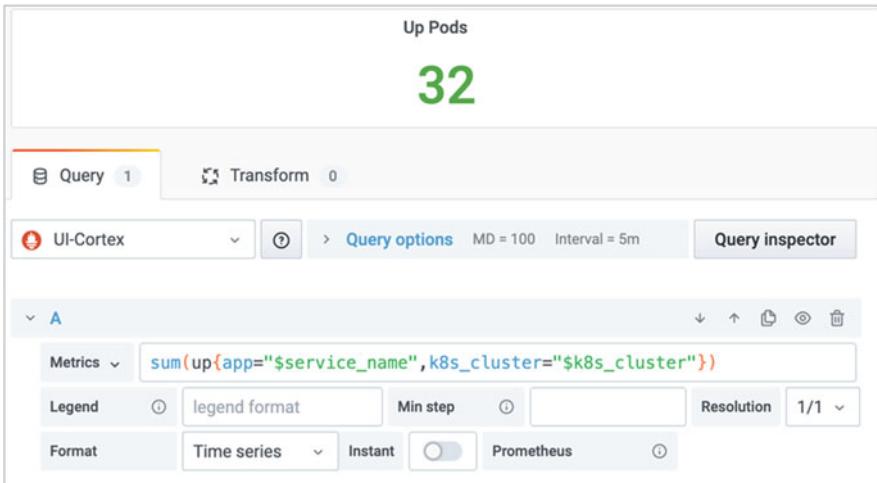


Fig. 7.30 Adding Stat Panel to show the number of running Pods

microservices, respectively, Fig. 7.29 shows the interface when adding service names.

After the addition is complete, two drop-down lists will appear in the dashboard showing all clusters and service names in the system, and you can choose to view the status of any microservice in a specific cluster, and then we can add the panel. We use the Stat Panel to show the number of running Pods, as shown in Fig. 7.30. The difference here from the normal panel is that you need to use variables in PromQL, such as `sum(up{app=\"$service_name\",k8s_cluster=\"$k8s_cluster\"})`, so that the panel will change dynamically as the variables change.

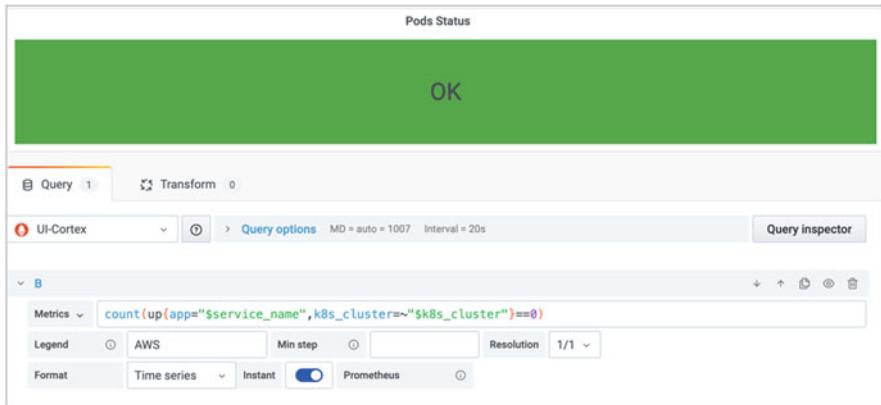


Fig. 7.31 Setting up the Pod Status Panel

Pod-specific information is presented through the Status Panel, and Fig. 7.31 shows how to set up the Pod Status Panel.

The PromQL content is as follows:

```
// The number of Pod in the cluster where a service is not running properly
count(up{app=\"$service_name\",k8s_cluster=~\"$k8s_cluster\"}==0)
```

This query counts certain service's pod numbers in the cluster that are not running properly. In the options section of the panel on the right side of the page, we set the panel to display the Warning status when the return value is greater than 1 and to display the Critical status when the return value is greater than 3, as shown in Fig. 7.32.

In this way, we have created a standardized metrics dashboard that allows you to view the real-time status of a particular microservice by selecting different clusters and services.

Build microservices standardized logging dashboard on Kibana

All microservices within the application have adopted the request log described in Sect. 7.2, which provides the basis for building a standardized logging dashboard for each microservice. Figure 7.33 shows the standardized logging dashboard for a particular microservice.

Fig. 7.32 Setting the threshold

Threshold	
Warning	1
Critical	3
Display Alias	Warning / Critical
Display Value	When Alias Displayed

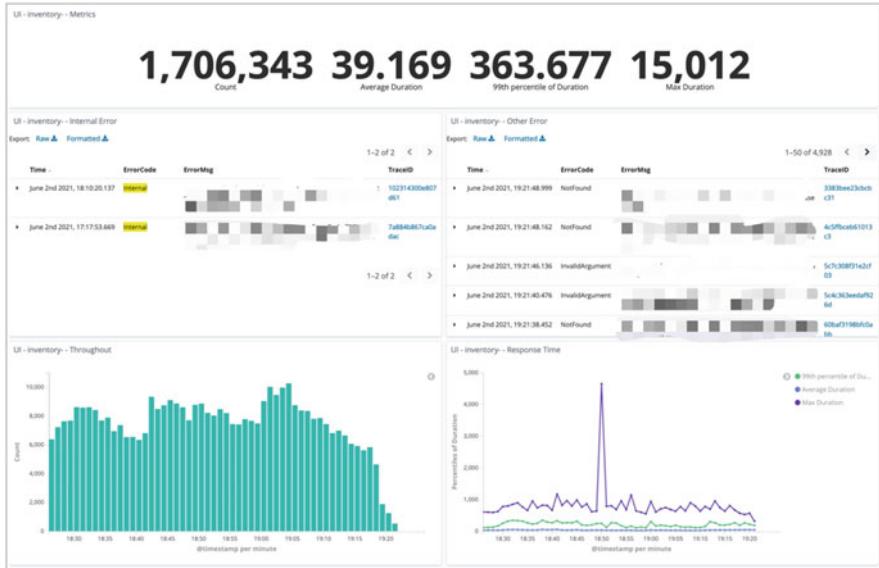


Fig. 7.33 Standardized logging dashboard

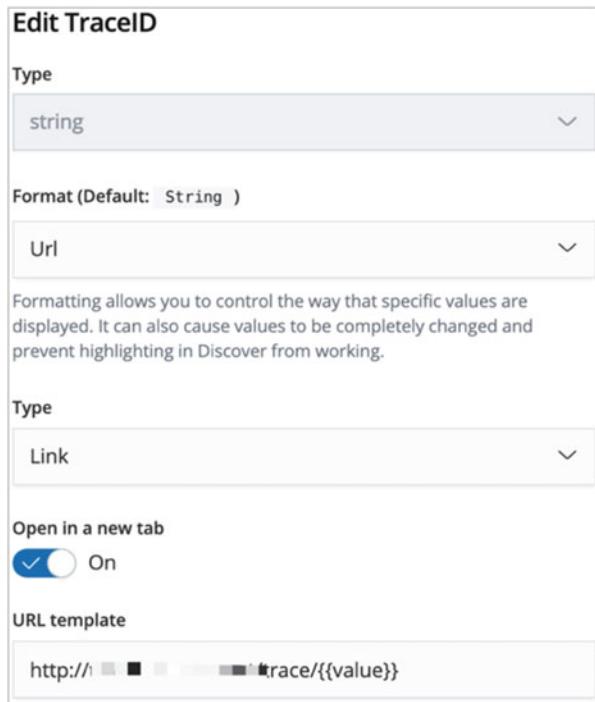
At the top of the dashboard are some key metrics about the service status, mainly including the number of requests processed, request latency, etc., followed by the error request information, and at the bottom are the service throughput and latency details.

We will introduce the error request information in detail here. Logs can carry rich information to help us debug problems. Here, we divide errors into two major categories and put them into two visualizations: one is internal errors, usually unanticipated runtime errors, such as connection problems, null pointer exceptions, etc. These errors are usually more severe and are defects of the service, which need to be handled or modified in code in time. Other errors are generally predictable, such as input parameter errors, object not found, etc. We can identify these kinds of improper calls from customers by proactive monitoring these errors and then notifying them to fix them.

In both visualizations, we display the span ID of the distributed trace directly and link to the distributed trace system, allowing you to quickly locate the problem by viewing the invocation chain of the erroneous request across the business system with a single click.

How is this done? First, the request log contains the span ID, and then we need to turn the span ID into a directly clickable link in Kibana's index pattern. As shown in Fig. 7.34, the default format of this field is a string, which we need to change to a link and add the address of the distributed tracing system to the URL template. Finally, the span ID is

Fig. 7.34 Linking log to tracing system



included as a column in the table when creating the error log visualization.

We will not introduce the detailed step of building each visualization on the dashboard here, because we just used several basic chart types provided by Kibana. However, as we know from the introduction in Sect. 7.2, the Kibana dashboard consists of a series of visualizations, each of which uses some index or search result as the data source. If we were to manually build this dashboard for all the microservices in our business system, it would be huge of effort, so we use a solution to automatically create dashboards through the Kibana API.

We need to create a standardized dashboard manually first and then export the dashboard to a file using the export function. Then, we can get the JSON descriptions of all the objects in this dashboard, and, finally, we can generate the service dashboard by replacing the necessary fields with the properties of the service.

Since we use a unified request log-based index pattern, we only need two parameters to create a standardized dashboard for a service: service name and index ID. According to the manual creation order, we start by creating a search object, then we use the search object to create visual charts (using the system internal error visual chart as an example), and, finally, we combine all the visual charts to form the dashboard. The code is shown as follows:

```

// Kibana API base address
const baseURL = 'http://elk-url/api/saved_objects'
// Generate a search object structure based on the service name and the corresponding request log index ID
const generateInternalErrorSearchBody = (serviceName, indexID) => {
  return {
    "attributes": {
      // Generate the title of the search object based on the service name passed in
      "title": `UI - ${serviceName} - Internal Error`,
      "description": "",
      "hits": 0,
      "columns": [
        "ErrorCode",
        ...
      ],
      "sort": [
        "@timestamp",
        "desc"
      ],
      "kibanaSavedObjectMeta": {
        "searchSourceJSON": `{"index": "${indexID}"}` // highlightAll":true,...}
      } // Create search criteria using the passed-in index IDs
    }
  }

// Create a search object by calling the API using the generated search object structure
// return the generated index ID for subsequent use in creating visual charts
const generateInternalErrorSearch = async (serviceName, indexID) => {
  // generate the structure of the search object
  const requestBody = generateInternalErrorSearchBody(serviceName, indexID);
  let id = ""
  // The Kibana API opens up different nodes for each object, the node to create the search object is 'search'
  await axios.post(`${baseURL}/search`, requestBody)
    .then(res => {
      id = res.data.id
    })
    .catch(...)
  return id;
}

// Generate the structure of the visual chart based on the service name and the index ID of the search object created in the previous step
const generateInternalErrorBody = (serviceName, searchID) => {
  return {
    "attributes": {
      // Generate the configuration information of the visual chart using the passed in service name and the search object's indexID to generate the configuration information for the visual chart
      "title": `UI - ${serviceName} - Internal Error`,
      "visState": `{"title": "${serviceName} - Internal Error", "type": "histogram", "params": {"uiStateJSON": "{}", "description": "", "savedSearchId": ${searchID}, "kibanaSavedObjectMeta": {"searchSourceJSON": "{}"}, "uiStateJSON": "{}"}, "version": 1}` // "version": 1
    }
  }
}

// Call the API to create a visual chart using the generated visual chart structure
// Return the generated chart ID for subsequent dashboard creation
const generateInternalError = async (serviceName, searchID) => {
  // Generate the visualization chart structure
  const requestBody = generateInternalErrorBody(serviceName, searchID);
  let id = ""
  await axios.post(`${baseURL}/visualization`, requestBody)
    .then(res => {
      id = res.data.id
    })
    .catch(...)
  return id;
}

// Compose all generated visual charts into a dashboard structure
const generateDashBoardBody = (serviceName, internalErrorID, ...) => {
  return {
    "attributes": {
      "title": `UI - Service - ${serviceName}`,
      "description": ""
    }
  }
}

```

```

// Generates the dashboard object structure based on all the visual chart IDs passed in
"panelsJSON": [
  {"\\"embeddableConfig\":{\\"vis\\\":[{\\"legendOpen\":false}],\\\"gridData\\\":{\\"h\\\":15,\\"w\\\":1,\\"x\\\":24,\\"y\\\":37},\\\"id\\\":\$throughoutID},\\\"panelID\\\":\$internalErrorID,...}, ...
  }
}
}

// Use the generated dashboard structure to call the API to create the dashboard
// Here we use meaningful, specified IDs instead of random IDs generated by Kibana
// This makes it easy to find the standardized template based on the service name
const generateDashBoard= async (serviceName, internalErrorID ...) => {
  const requestBody = generateDashBoardBody(serviceName, internalErrorID ...);
  let id = ""
  await axios.post(`${baseURL}/dashboard/ui-service-$service{serviceName}-standard`, requestBody)
    .then(res => {
      id = res.data.id
    }).catch(...)
  return id;
}

// Generate the overall system status dashboard by calling "search object - visual chart - dashboard"
const generateStandDashboard = async (serviceName, indexID) => {
  const internalErrorSearchID = await generateInternalErrorSearch(serviceName, indexID);
  const internalErrorID = await generateInternalError(serviceName, internalErrorSearchID);

  // ...
  const id = await generateDashBoard(serviceName, internalErrorSearchID ...);
  return id;
}

// The services that need to generate a standardized dashboard and the corresponding index ID
const allServices = [{serviceName: 'inventory', indexID: '1242f1f0-d601-11ea-b7fc-7522b4f7be57'}, ...];
// Loop through the dashboard generation methods to generate a standardized dashboard for all services
allServices.forEach((svc) => (generateStandDashboard(svc['serviceName'], svc['indexID'])));

```

(b) Dashboard to show overall system status

As shown in Fig. 7.35, we built a dashboard on Grafana that shows the current status of all services; each service has links to its two standardized dashboards and the distributed tracing system, so that you can see the real-

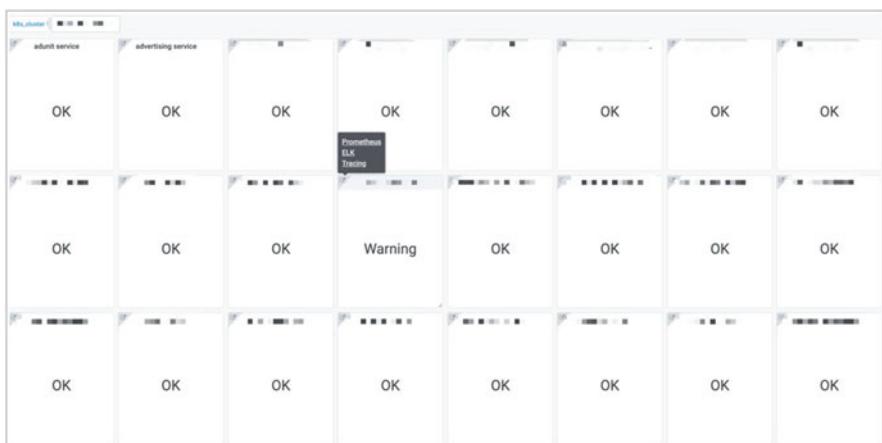


Fig. 7.35 Dashboard to show overall system status

time status of all services through a unified portal and easily access metrics, log data, and the distributed tracing system.

Here, we also use Grafana's dashboard template and also use the “Repeat by Variable” feature so that we only need to create one dashboard and then iterate through all the service names to create status panels for all the services, thus avoiding duplication of work. We reuse the status shown on the standardized dashboard for a single service to show the status of the services on the current dashboard, i.e., Pod, CPU, memory, and error rate metrics, and an exception in any of them will cause the status panel of a service to change to warning or critical status.

As for the standardized dashboard for individual services, the creation of the overall system status dashboard begins with the creation of two custom variables: the cluster name and the service name. The difference is that the service name should be a multi-select option and should be set to hidden.

Create a status panel with four PromQLs (corresponding to the four status panels of a single service dashboard, noting the use of variables), as shown in Fig. 7.36. Then, set separate thresholds and links for each metric, consistent with the panels in the standardized metrics dashboard.

Add three links to the standardized dashboards and the tracing system, again using variable values to generate the links here. Then, set Repeat based on the service name in the Repeat options section (container_name in Fig. 7.37), then our overall system status dashboard is created.

2. Business Monitoring

Generic microservice monitoring focuses on monitoring the health of the system to identify and resolve issues on time. In addition, each service can also build custom dashboards to count and analyze the usage of the service according to its own business. Our business system is mainly for B2B business, so customer-based or function-based monitoring is the most common scenario, such as monitoring the usage of a module or function, the usage of the system by a certain customer, etc.

Figure 7.38 shows the business monitoring dashboard for a service of the system, divided into two parts, left and right, which count the usage of the business system by two types of customers. The left side is the customers with the publisher role and the right side is the customers with the buyer role. We have counted the total number of calls, latency, and the distribution of different customers calling the system for these two types of customers.

The monitoring system can help us proactively identify problems, find the causes, and finally solve them. However, we can't 7*24 keep an eye on the monitoring system, so we also need to build an alert system to notify proper persons in time when the problem occurs.

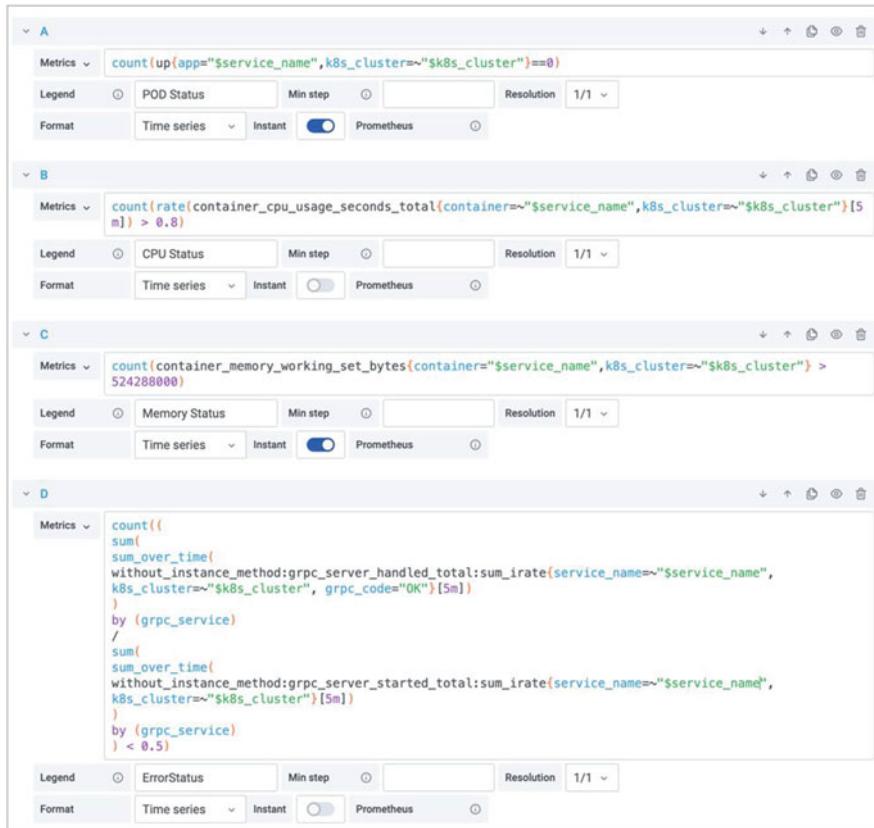


Fig. 7.36 Setting up status panel

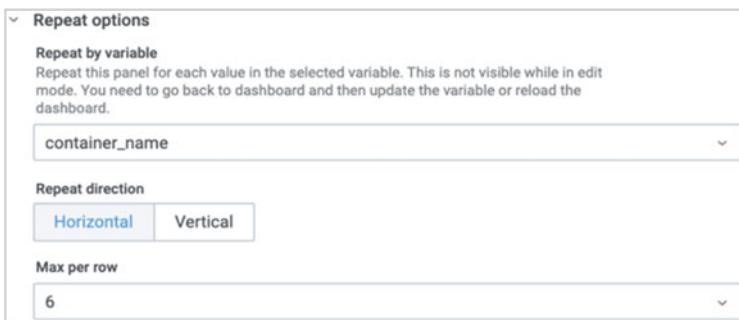


Fig. 7.37 Setting repeat options

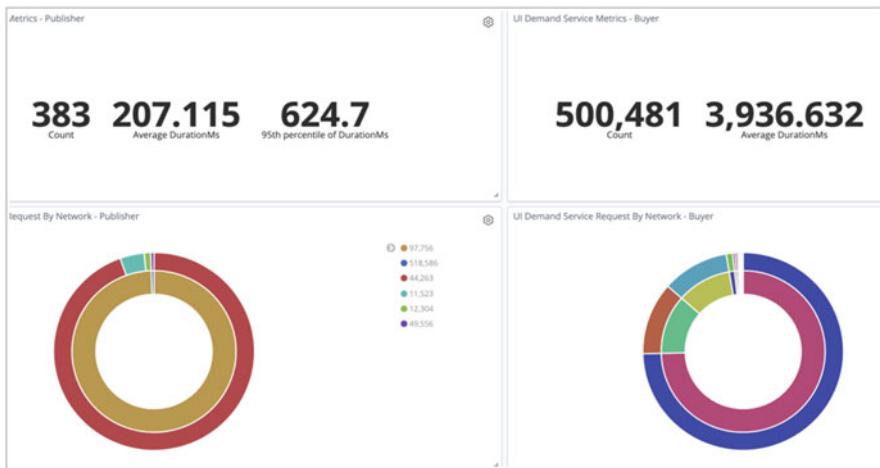


Fig. 7.38 A business monitoring dashboard

7.5.2 Alert System

After presenting the three pillars of observability in 2017, Peter Bourgon discussed the deeper meaning of metrics, tracing, and logging in engineering at GopherCon EU 2018, where the sensitivity of anomalies was mentioned: metrics are the most sensitive to anomalies, logging is second, and tracing is most often used in troubleshooting and locating the problem. Therefore, our alerting system is mainly built on top of metric and log data. Figure 7.39 shows the architecture of the alerting system built by our team.

We will present its parts in a hierarchy below.

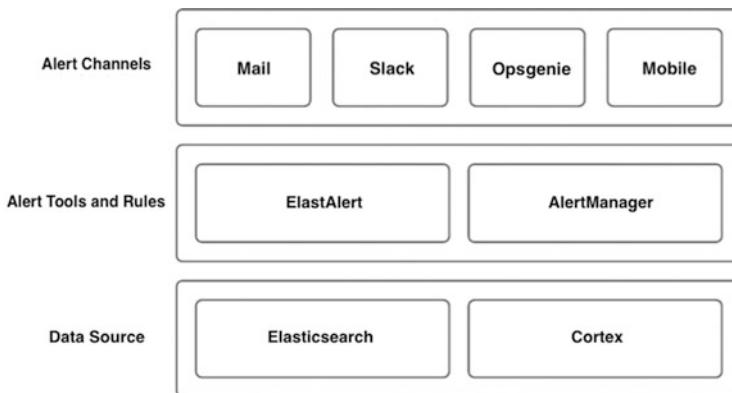


Fig. 7.39 The architecture of the alerting system

1. Layers of the alert system

We have divided the alert system into three layers from the bottom up, as follows.

The first layer is the data source, where log and metrics data are collected, processed, and stored into Elasticsearch and Cortex, respectively.

The second layer is the alert tools and rules, as follows:

- (a) The monitoring platform infrastructure support team developed AlertManager as an alerting tool for metric data and ElastAlert for alerts based on log data.
- (b) Define standard and unified alerting rules, which are the basis for all services. Each service must build these alerting rules, which focus on four golden signals.
 - (i) Traffic: Abnormal traffic alert, for example, triggers an alert if the number of requests per minute exceeds 1000.
 - (ii) Latency: Slow request alert, for example, trigger alert when the median response time is greater than 1 s within 5 min.
 - (iii) Error, request error alert, for example, if the number of error requests exceeds 20 in 3 min, the alert is triggered.
 - (iv) Saturation: Mainly refers to the system resource abnormality alert, for example, the alert is triggered if the CPU usage of Pod exceeds 90%.
- (c) In addition to the standard alert rules, services can also customize alert thresholds based on their own business.

The third layer is the alerting channel, which supports email, Slack messages, Opsgenie, and mobile messages (both SMS and phone).

We divide the alerts into three levels, and each level corresponds to a different alert channel and processing rules.

- (a) Info: Notifies only the service maintainer, usually sending emails and Slack messages.
- (b) Warning: In addition to notifying the service maintainer, the company's Network Operations Center (NOC) team will be notified, and NOC engineers will notify the development engineer team to handle these alerts timely.
- (c) Critical: In addition to normal emails and Slack messages, Opsgenie's hierarchical alerting mechanism is triggered by sending emails to specific addresses in Opsgenie to ensure that alerts are handled on time.
 - (i) Opsgenie will be the first to notify service maintainers via SMS.
 - (ii) If the alert is not handled within 3 min, a call will be made to the service maintainer.
 - (iii) If an alert is not handled within 5 min, the development manager will be notified directly.

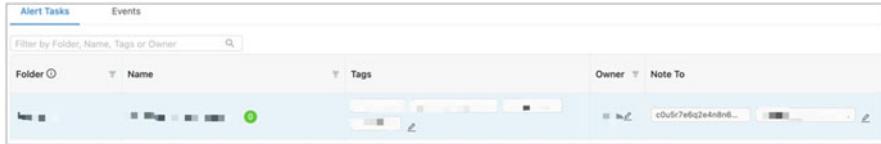


Fig. 7.40 Alert Manager overview

2. Use AlertManager to build alerts based on metric data

AlertManager is a powerful, metric-based alert management platform developed by the company's monitoring platform infrastructure support team. In addition to providing a graphical interface to help us create flexible alert rules, it also manages alert rules for each team in the company, such as folders, tags, and other rules, as shown in Fig. 7.40. It also supports Acknowledge Receipt (Ack) operation for alerts that have been triggered, as well as muting alerts for a certain period when needed (e.g., during system maintenance). This section will not focus on the functions of this alert management platform but will only briefly introduce how to set alert rules.

Metric-based alerts mainly focus on some important indicators, such as those related to system resources. We briefly describe the alert settings for a service Pod abnormal state. Figure 7.41 shows some basic settings of alert rules, such as rule name, storage folder, data source settings, etc. AlertManager supports various types of alert tasks, and here we only take standard alerts as an example.

Figure 7.42 shows how to set alert rules, including how to query the number of Pods with abnormal status, as well as the settings of alert threshold and level, where we set a warning alert to be triggered when the returned data is greater than

The screenshot shows the 'Alert Setting Page' with the title 'General Configuration'. The page contains several input fields and dropdown menus:

- * Folder: A dropdown menu with several icons.
- * Name: An input field with a dropdown arrow.
- * Owner: An input field with a dropdown arrow.
- * Receivers: A dropdown menu containing many small, dark square icons.
- * Notify NoData: A dropdown menu with options: NONE (selected), Email, and Post Action.
- * Post Action: A dropdown menu with options: NONE (selected), Email, and Post Action.
- Also send alerts to Owner(ycao's notify email is empty)
- Required Tags:
 - * Team: A dropdown menu with one icon.
 - * Component: A dropdown menu with 'General' selected.
- Custom Tags: A dropdown menu with several small square icons.
- * Data Source: A dropdown menu with 'UI-Cortex' selected.
- * Task type: A dropdown menu with three options: Simple (selected), Standard, and Wave(beta).

Fig. 7.41 Basic settings of alert

The screenshot shows the configuration interface for an alert rule. Under 'Alert conditions', 'INFO' and 'CRITICAL' are selected. For 'INFO', there is no condition. For 'WARNING', the condition is 'UI-Cortex::kube_pod_status_ready > 2'. For 'CRITICAL', the condition is 'UI-Cortex::kube_pod_status_ready > 4'. Under 'Alert grouping & Scope', it says 'of each①: pod x'. Below this, it says 'I want to monitor data with the following conditions:' and lists 'namespace IN [false x unknown x]', 'condition IN false x unknown x', 'pod MATCH [multiple icons]', and a final row starting with '① - +'. There are also some circular icons with minus and plus signs.

Fig. 7.42 Setting up alert rules (edited)

2, and a critical alert to be triggered when it is greater than 4. In addition to this, you can also set the message body of the alert.

3. Use ElastAlert to create alerts based on log data

ElastAlert is an open-source alerting system based on Elasticsearch developed by Yelp. The main function is to query data matching predefined rules from Elasticsearch for alerting. The log data of our business systems are stored in Elasticsearch after processing, which is one of the main reasons we choose ElastAlert.

ElastAlert supports 11 types of alert rules, and the following are a few common rule types.

- (a) Frequency: X events in Y period.
- (b) Flatline: The number of events X in Y period is less than a certain value.
- (c) Any: Match any that matches the filter.
- (d) Spike: An increase or decrease in the incidence of matching events.

Here, we will not introduce the details of each alert rule, only take the most commonly used rule type—Frequency as an example to briefly introduce the ElastAlert alert rule settings; part of the code is as follows:

```

# Rule name, must be unique
name: xxxx Service 500 Error Frequency Alert - 5m

# Rule type
# The frequency rule will trigger an alert if the number of occurrences of an event in a certain time window exceeds the defined threshold
type: frequency

# Index to be queried, supports wildcard
index: fw-xxxx-request-log-* 

# The number of events queried in the custom time window will trigger an alert if it exceeds this value
num_events: 10

# Time window
timeframe:
minutes: 5

buffer_time:
minutes: 5

# Filter conditions to query Elasticserach, the relation between conditions is 'OR'
filter:
- query:
  query_string:
    query: "ErrorCode:\"Internal\" OR ErrorCode:\"Unknown\" OR ErrorCode:\"Unavailable\" OR ErrorCode:\"DeadlineExceeded\""

# Alert channel
alert:
- "email"

# Alert subject, using a templated title, the placeholder "@{0}" will be replaced by the timestamp field of the record returned
alert_subject: "[FAIL|PRD] xxxx Service 500 Errors @{0}[ElastAlert]"
alert_subject_args: ["@timestamp"]

# The alert message body, including the steps to take when an alert is received
alert_text: "Action Items:\n\nFor NOC, please call xxxx UI ENG.\n\nFor ENG, please check http://elk.fwmmr.net/app/kibana#/discover/5a740840-c028-11ea-8198-1d52d5d30c9c?_g=(time:(from:now-7d,mode:quick,to:now))\n\nOwner: @xxxx\n"
# These fields will appear in the alert message
include: ["NetworkID", "ErrorCode", "ErrorMsg", "TraceID"]

# Alerts need to be sent to the following mailing list
email:
- "eng-xxxx@example.com" // Service maintainer, usually an email group
- "alerts@example.com" // NOC team
- "xxx@slack.com" // Unified Slack alert notification via email integration

```

7.6 Summary

In this chapter, we focus on the adoption of service observability in our business systems. In Sect. 7.1, we elaborate on the definition and use of observability in the cloud-native domain, introduce the three pillars of observability (metrics, logging, and tracing) and the current state of community products, and discuss the difference and relation between observability and traditional monitoring. Subsequent Sects. 7.2–7.4 describe the adoption of each of the three pillars in the system, which is the basis for making services observable. Finally, in Sect. 7.5, we describe how to organically correlate and use data from the three pillars of observability to enable efficient monitoring and alerting of business systems.

When developing microservice applications with observability, I suggest you choose community products reasonably according to the current situation and needs of your own products and finally build an observable system that suits your system.

Chapter 8

Quality Assurance Practices



For applications based on a microservice architecture, the large amount of interservice communication leads to increased system complexity, and it becomes more difficult to build a complete and usable test environment. At the same time, the long invocation links also make debugging inconvenient, making it difficult to test complex systems through traditional methods.

Over the past few years, our team has transformed a monolithic application into microservices-based architecture. In this chapter, we will focus on quality assurance practices during architecture migration and talk about how the team is building a quality assurance system in the cloud-native era through sound testing techniques and chaos engineering.

8.1 Quality Assurance System

The Waterfall Model places software testing and maintenance activities after program writing, resulting in software product quality assurance practices becoming the last gatekeeper before software product delivery. For products with rapidly changing requirements, this is neither conducive to parallel development to accelerate the project schedule nor to identify problems early in the project. This is because errors in the software development process are transmissible; i.e., if an error in the requirements analysis phase is not identified, it is sequentially passed on to subsequent phases such as product design and coding and testing. Therefore, the earlier the quality assurance effort enters the software development cycle, the lower the relative cost required to identify and resolve problems.

Our team has explored and built a quality assurance system for microservice architecture in the past few years of agile development practice. By introducing quality assurance practices in each life stage of the application, we have greatly reduced the possibility of reworking in later stages while setting up corresponding acceptance criteria at each stage so that each stage can follow the same rules.

8.1.1 *Quality Challenges*

Microservice architecture changes present new challenges for quality assurance, mainly in the following areas.

1. Significant increase in interactions

The transition from a monolithic application architecture to a microservice architecture requires splitting a large application into several small, highly cohesive, and low-coupling services, which are deployed and run independently and communicate with each other through a contract defined by an interface. This allows the entire application to be split into smaller modules for development, reducing development and maintenance costs and increasing the flexibility and scalability of the service itself. However, the architecture topology of the whole system becomes more complex. How to delineate the boundaries between microservices and design reasonable upstream and downstream invocation relationships has become a problem that architecture designers have to think about.

2. Service componentization

The componentized nature of microservices also puts forward new requirements on the team's parallel development collaboration process in the following three areas.

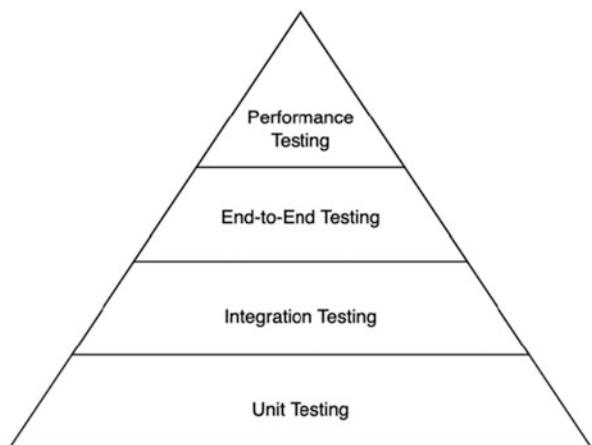
- (a) Increased interservice communication costs: The number of communications between teams maintaining different microservices increases significantly, especially when there are different delivery cycles for different microservices. At this point, if there is a communication problem, the delivery of the entire application that goes live can be affected.
- (b) The increased cost of interservice collaboration: The increased interaction within the system makes it more expensive to build a complete test environment. This also means that you need to wait until all sub-services are developed before you can perform the overall interworking, which puts forward a higher requirement on the schedule control of the whole system development.
- (c) Debugging operation cost and maintenance efforts increase: When problems arise after the system is cascaded or goes live and needs to be troubleshooted, it often requires the association of multiple members of the microservice maintenance team to solve them, which invariably increases human and material costs.

In addition, the componentization of microservices changes the invocation from in-process to out-of-process, and the original local transaction becomes a distributed transaction. To ensure the reliability of the system, we need to introduce measures such as timeout retries. These have been described in the previous chapters and are omitted here.

3. Traditional testing methods are no longer applicable

Testing methods commonly used in monolithic application architectures are not necessarily applicable to microservice architectures. For example, in a

Fig. 8.1 A four-layer structured testing pyramid



monolithic application architecture, testers often only need to use end-to-end testing to verify the functionality of the overall system. In a microservice architecture, the testers are mostly from different microservice teams and often lack a comprehensive understanding of the overall system, so it is necessary to adjust the testing strategy, not only to verify whether the whole link works properly but also to ensure that the interfaces of each microservice are implemented as expected.

8.1.2 *Testing Strategy*

The new challenges introduced by the microservices architecture have put forward new requirements on testers, and our team has concluded new testing strategies during the architecture migration process, specifically in the following categories.

1. Test Layering

Mike Cohn's Test Pyramid reveals the idea of testing layering, but, because there are so few layers, it is somewhat powerless in the face of microservice architectures. The idea of layering for microservice architecture is as follows:

Write tests with different granularity for different layers.

The higher the level, the lower the number of tests it should have.

This theory can be used as a guide to build a test pyramid suitable for our own team. Our team finalized a four-layer structured testing pyramid model for daily business development based on the actual situation; see Fig. 8.1.

From the bottom to the top in order: unit testing, integration testing, end-to-end testing, and performance testing. Specific testing practices will be described in detail in the next section.

2. Test automation

The distributed nature of microservices themselves implies that the test cases are also distributed. Manual testing may be sufficient for testers of a single microservice, but, when the number of microservices increases dramatically, the need for automated testing becomes especially urgent for the entire team.

Under the microservice architecture, improving the coverage of automated testing puts forward new requirements on the testing environment, testing techniques, and testing methods, as follows:

- (a) For the test environment, you need to ensure that there is a dedicated person to maintain it to avoid not having the environment available when testing is needed and not knowing who to call for maintenance when there is a problem with the environment.
- (b) For testing techniques, we need to select the appropriate testing techniques based on the characteristics of different microservices. However, it should be noted that the diversity of technologies can lead to increased costs for test environment building and maintenance, and we recommend using the same set of testing technologies and programming languages within the team as much as possible if based on the same technology stack.
- (c) For testing methods, we need to adapt the testing methods previously used for monolithic applications and select different levels of testing methods for the characteristics of microservices to ensure delivery quality.

In addition, the smooth running of automated testing relies on building the infrastructure for continuous integration and continuous deployment, which can be found in Chap. 9 of this book and is not covered in detail here.

3. Progressive adjustments

For most teams, the evolution from monolithic to microservices architecture does not happen overnight, so the testing strategy needs to be adjusted incrementally to match the pace of the development team's architectural evolution. For example, for teams that are just starting to move to a microservice architecture, our recommended approach is to keep the upper layer interfaces intact so that the end-to-end testing of the original monolithic application can still work and to start writing unit and integration tests in the new microservice architecture to lay the foundation for future application development.

When a system is in different life stages, it has different quality objectives, and the testing strategy should be adjusted accordingly. For example, for systems that are in a hurry to go online, end-to-end testing can be done first to ensure the availability of external services, and then unit testing and integration testing can be done after the system is stable to pay off this part of the “testing debt.”

8.1.3 Building a Quality Assurance System

It is important to note that in Sect. 8.1.2 we mentioned the testing strategy used in practice, but testing is ultimately just one piece of the quality assurance system. What we need is a complete quality assurance system to guarantee the continued high quality of the business. In order to extend the concept of quality assurance to the whole product lifecycle, we need to make quality assurance concrete and standardized through technical measures and management processes and make it a code of conduct for daily testing, i.e., to create a quality assurance system suitable for the team itself. Before building a quality assurance system, we need to establish a quality assurance model.

1. Development of a quality assurance model

Paul Herzlich proposed the W model of software testing in 1993. Compared to the V model, the W model separates the testing behavior from the development behavior and allows both types of behaviors to be performed simultaneously, thus allowing testing to occur throughout the software development cycle. Based on this advantage, we borrowed the W model to build a quality assurance model, as shown in Fig. 8.2. Each phase of building this model is described in detail below.

- (a) *Requirements definition phase:* The product manager gives key test points of User Acceptance Testing (UAT, User Acceptance Testing) as criteria for application acceptance.
- (b) *Requirements analysis and system design phase:* We hold a Tech Approach meeting to discuss the advantages and disadvantages of several technical implementation options and determine the final technical solution. For requirements involving multiple microservices or multiple applications, we invite developers and testers from the corresponding teams to review the requirements together and create an interface document for all parties to “sign off.” This allows each team to develop and test their own application based on the interface document.

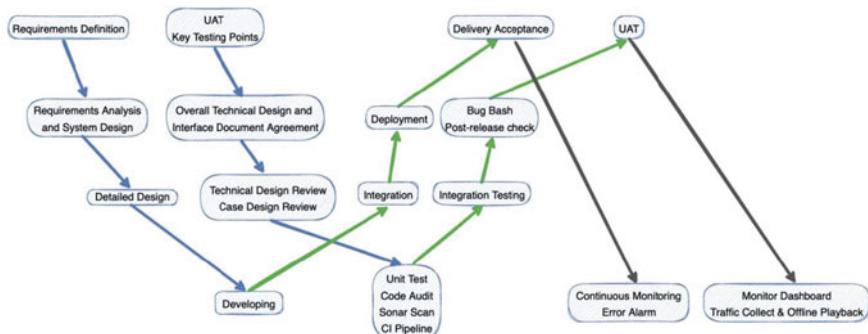


Fig. 8.2 The W model of software development cycle

- (c) *Detailed design phase:* We hold two types of meetings, technical design meetings and case design meetings, to discuss the technical details and test case design in the application implementation solution, respectively. For requirements involving multiple microservices or multiple applications, a special integration test case design meeting is also held. This meeting will again involve the developers and testers from the corresponding teams to confirm that the test cases meet the requirements and to agree on a future point in time for integration testing.
- (d) *Developing phase:* Developers and testers write business code and write unit tests and regression tests to ensure that the implementation is as expected. We use code audits, Sonar testing, and the results of the continuous integration pipeline to ensure the quality of the code at this stage through.
- (e) *Integration phase:* Testers execute the integration tests previously agreed upon by all parties through the built test platform.
- (f) *Deployment phase:* Developers and testers will then perform a bug Bash and Post-release Check online to ensure that the functionality works as expected after the feature go-live.
- (g) *Delivery acceptance phase:* The product manager will conduct user acceptance testing based on the key points identified in the previous UAT doc and deliver the product to the customer after completion while achieving blue-green deployment and canary release with the help of the continuous integration pipeline, making the delivery more flexible and controllable.
- (h) *Continuous monitoring error alarm phase:* Developers and testers will add corresponding monitoring alarms to observe the whole system. If necessary, online traffic will be collected and played back offline to meet the needs of offline debugging, big data testing, etc.

At various phases of the model, we may identify new problems and adjust the product design, which requires the product manager and technical manager to intervene in a timely manner to assess the risk of the application development situation and determine the next step. In this case, if the engineer's understanding of the requirements themselves does not deviate significantly, developers and testers will be able to go back to one of the previous phases to revise the design and modify the corresponding code.

2. Quality assurance system in practice

The focus of the QA model is primarily on the offline and online parts of a business from inception to delivery. In addition to this, we need to be aware that testing frameworks, continuous integration pipelines, etc. also play an important role. A strong quality culture within the team itself also contributes to quality assurance. Combining these, we built a complete quality assurance system, as shown in Fig. 8.3.

We have divided the quality assurance system into three main parts: offline, online, and infrastructure, which are described below.

The offline section deals with quality assurance-related operations during the development phase:

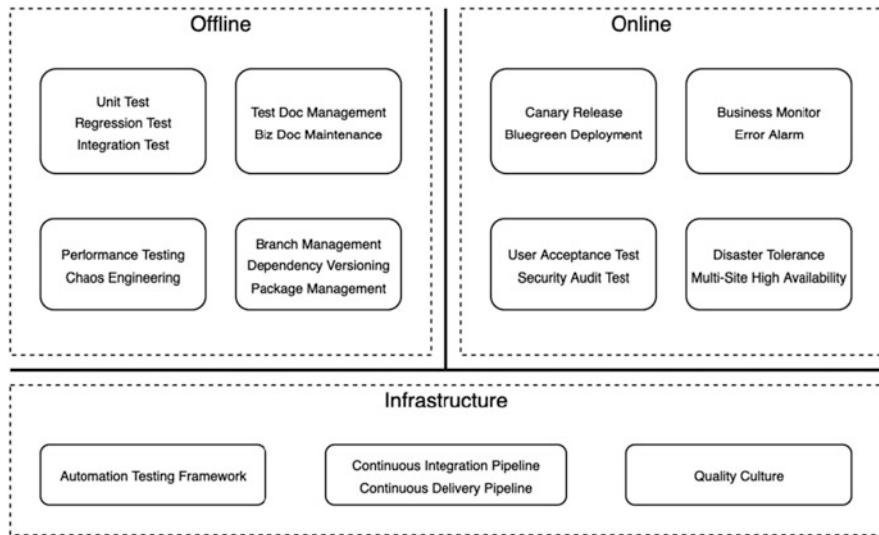


Fig. 8.3 Quality assurance system

- Basic testing practices: covering unit test, regression test, and integration test, which the author will cover in detail in the next section.
- Document management: Along with the architecture evolution and business adjustment, test documents need to be consolidated, and business documents need to be maintained in time.
- Performance testing and chaos engineering: Link-level performance testing helps the team to assess system bottlenecks and thus prepare for potential problems in advance. Chaos engineering, on the other hand, is a “failure exercise” in the form of randomized experiments. We can observe the system boundaries through the response of the system against chaos experiment.
- Branch management, dependency versioning, and package management: At the code deployment level, we need to maintain multiple environments for development, testing, pre-release, and production, thus testing needs to be done for different branches. On top of that, there is a need to manage different versions of dependencies and installed packages to ensure that they work properly in each environment.

The online section deals with quality assurance related operations that are performed after the product is released to the production environment.

- Canary Release and Bluegreen Deployment: essential skills for cloud-native applications; see Chap. 9.
- Business monitoring and error alerting: see the practices on system observability in Chap. 7 of this book, which will not be repeated here.
- User acceptance testing and security audit testing: UAT was described earlier when the testing model was introduced. Security audit testing will ensure that

there are regular auditors reviewing the security policies of the software used in the system to ensure that there are no vulnerabilities or risk exposures.

- (d) Disaster tolerance and multi-site high availability testing: applications in the cloud-native era still require disaster tolerance and multi-site high availability to ensure the availability and reliability of the system.

Infrastructure is used to facilitate the implementation of quality assurance systems, such as testing tools, quality concepts, etc., and is divided into three categories.

- (a) Automation testing framework: An automation testing framework helps simplify the difficulty of writing test code at all levels for team members while also providing key data such as code coverage more quickly.
- (b) Continuous Integration and Continuous Delivery Pipeline: Continuous Integration and Continuous Delivery support enable faster iteration of microservices systems and combined with automated testing can help teams find problems and resolve them faster.
- (c) Quality culture: With the rapid iteration of systems in microservices architecture, quality assurance cannot be achieved without the efforts of every member of the team. Therefore, it is very important to build a quality culture in the team. In the next sections, we introduce some practices such as Quality Issue Sharing Sessions and Bug Bash (Bug Sweep). Based on this, it is possible to build a top-down quality culture by popularizing quality concepts and participating in person.

8.2 Testing Practices

In the testing pyramid, from the bottom to the top are unit testing, integration testing, end-to-end testing, and performance testing. The closer to the bottom, the faster the testing, the shorter the feedback cycle, and the easier it is to locate the affected functions after testing finds problems; the closer to the top of the pyramid, the larger scope the test covers, the longer it takes to complete the testing, and the more guaranteed the correctness of the functionalities after testing.

In this section, we will specifically cover unit testing, integration testing, and end-to-end testing practices, as well as testing automation.

8.2.1 Unit Testing and Mock Practice

A “unit” is the smallest testable part of the software. Unit testing is the first and most important part of software development where the smallest components are tested for

correctness, and it is the bottom of all tests, written by developers in parallel with the code development. In this session, we will cover the ground rules, table-driven testing, mock practices, and frequently asked questions.

1. Ground rules

The Go language has built-in support for unit testing and benchmarking, which is lightweight and easy to be used with the ‘go test’ command to write and execute tests. The testing framework defines some rules for test files:

- (a) Each test file must be named with the *_test.go* suffix, usually in the same package as the file being tested.
- (b) The function name of the unit test must start with *Test*, as an exportable function.
- (c) The unit test function must take a pointer to a *testing.T* type as an argument at definition time and has no return value.

For example, the test function is named *GenDatesSlice* and the corresponding function is defined as follows:

```
func TestGenDatesSlice(t *testing.T)
```

2. Table-driven testing

Golang officially recommends the table-driven approach to writing unit tests, which involves repeatedly specifying a few inputs and outputs for different use cases. By listing the inputs and outputs in a table, the tests are executed by iterating through them.

For example, as a maintainer of the following utility function code, this way is straightforward. Also, appending a new test case to it is only a matter of adding a row to the table:

```
// Round implements rounding of floating point numbers
func Round(f float64, decimals int) float64 {
    d := math.Pow10(decimals)
    if f < 0 {
        return math.Ceil(f*d-0.5) / d
    }
    return math.Floor(f*d+0.5) / d
}
```

The corresponding table-driven test cases are as follows:

```

func TestRound(t *testing.T) {
    type args struct {
        f      float64
        decimals int
    }

    tests := []struct {
        name string
        args args
        want float64
    }{
        {"0.000", args{0.000, 2}, 0.00},
        {"0.005", args{0.005, 2}, 0.01},
        {"-0.005", args{-0.005, 2}, -0.01},
        {"1.555", args{1.555, 2}, 1.56},
        {"-1.555", args{1.555, 2}, -1.56},
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            if got := Round(tt.args.f, tt.args.decimals); got != tt.want {
                t.Errorf("Round() = %v, want %v", got, tt.want)
            }
        })
    }
}

```

3. Mock practice

There is often a requirement for independence when writing unit tests. Also, usually due to the complexity of the business logic, the code logic also becomes complex and dependent on many other components, resulting in more complex dependencies when writing unit tests, such as the database environment, network environment, etc. Thus, the workload of writing unit tests is greatly increased.

The main idea to solve this type of problem is to use mock. Two unit test mock practices exist in our team. One is the testify/mock practice used in conjunction with the mockery command, and the other is the gomock practice based on the mockgen command, which we describe separately below.

(a) Testify/mock

Testify provides an excellent Golang testing toolkit that includes support for assertions, mocks, and other common testing requirements. However, when writing unit tests, there are often many methods and functions in the application's interface that need to be mocked, and writing mock code one by one would be a developer's nightmare. At this point, we choose to take advantage of the ability to quickly generate code provided by mockery. Mockery can find all interfaces in a specified directory and then automatically generate mock objects based on testify/mock in the specified folder.

For example, for the interface of:

```

type NetworkDomain interface {
    GetNetworkByld(networkId int64) (*mdproto.Network, error)
}

func (domain *networkDomain) GetNetworkByld(networkId int64) (*mdproto.Network, error) {
    // ...
    return response, nil
}

```

The code generated by calling mockery is as follows:

```

type NetworkDomain struct {
    mock.Mock
}

// GetNetworkByld provides the mock method
func (_m *NetworkDomain) GetNetworkByld(networkId int64) (*proto.Network, error) {
    ret := _m.Called(networkId)
    // ...
    return r0, r1
}

```

For the following files to be tested:

```

type dataRightDomain struct {
    networkDomain NetworkDomain
    // Use NetworkDomain in mock code
}
func (domain *dataRightDomain) GetDataRightWhitelist(all bool, searchQuery *types.SearchQuery) ([]*business.WhitelistItem, int32, error) {
    // ...
    partner, err := domain.networkDomain.GetNetworkByld(item.Id)
    // Get the return value from the mock function when using mock
    // ...
}

```

Its test file is below:

```

func TestGetDataRightWhitelist(t *testing.T) {
    // initialize mock
    networkDomainMock := & mock.NetworkDomain {}
    // Set the expected return value of the mock function
    networkDomainMock.On("GetNetworkByld", mock2.Anything).Return(nwRet, nil)
    dataRightDomain.networkDomain = networkDomainMock

    // Call GetDataRightWhitelist, the networkDomain of dataRightDomain has been replaced with networkDomainMock
    witems, number, err := dataRightDomain.GetDataRightWhitelist(true, searchQuery)

    // ...
}

```

(b) gomock

gomock is an official Golang-maintained testing framework that generates “.go” files containing mock objects via the self-contained mockgen command. It provides two modes to support the generation of mock objects: source file mode and reflection mode.

If source file mode is used, you can specify the interface source file with the -source parameter, the generated file path with -destination, and the package name of the generated file with -package, as shown in the example below:

```
mockgen -destination foo/mock_foo.go -package foo -source foo/foo.go
```

If no interface file is specified with -source, mockgen also supports the generation of mock objects by reflection, which takes effect via two non-flagged arguments: the import path and a comma-separated list of interface symbols. An example is shown below:

```
mockgen database/sql/driver Conn,Driver
```

In addition, if there are multiple files scattered in different locations, to avoid executing the mockgen command multiple times, mockgen provides a way to generate mock files by annotating them, which is done with the help of Golang’s own go generate tool. For example, add the following comment to the interface file:

```
//go:generate mockgen -source=foo.go -destination=. /gomocks/foo.go -package=gomocks
```

One advantage of this approach is that for teams that are still exploring mock practices, the mock files can be added incrementally in this way, thus reducing the cost and risk of switching code to different mock solution if needed.

For example, for interfaces that nest other interfaces like the following:

```
type AvailableTelevisionNetworkBIO interface {
    // metadata service is a dependency, put its interface here
    grpc.MetadataNetworkServiceInterface

    List(networkID, userID int64, req *proto.ListAvailableTelevisionNetworksRequest) ([]*TelevisionNetwork, error)
}
```

The mock file generated by calling mockgen is as follows:

```
// MockAvailableTelevisionNetworkBIO is an implementation of the AvailableTelevisionNetworkBIO interface
type MockAvailableTelevisionNetworkBIO struct {
    ctrl *gomock.Controller
    recorder *MockAvailableTelevisionNetworkBIORecorder
}

// ...

// mock implementation of the List method
func (m *MockAvailableTelevisionNetworkBIO) List(networkID, userID int64, req *proto.ListAvailableTelevisionNetworksRequest) ([]*bio.TelevisionNetwork, error) {
    m.ctrl.T.Helper()
    ret := m.ctrl.Call(m, "List", networkID, userID, req)
    ret0, _ := ret[0].([]*bio.TelevisionNetwork)
    ret1, _ := ret[1].(error)
    return ret0, ret1
}
```

For the following functions to be tested, the:

```
// AvailableTelevisionNetworkDomainImpl implements the Domain layer functions
type AvailableTelevisionNetworkDomainImpl struct {
    bio bio.AvailableTelevisionNetworkBIO
}

// List returns the available television records for the current client
func (a *AvailableTelevisionNetworkDomainImpl) List(networkID, userID int64, req *proto.ListAvailableTelevisionNetworksRequest) (*proto.ListAvailableTelevisionNetworksReply, error) {
    // ...
    availTVNetworks, err := a.bio.List(networkID, userID, req)
    if err != nil {
        return nil, err
    }
    // ...
}
```

Its test function is as follows:

```

func TestAvailableTelevisionNetworkDomain_List(t *testing.T) {
    // ...
    m := NewMockAvailableTelevisionNetworkBIO(ctrl)
    m.
        EXPECT().
        List(
            gomock.AssignableToTypeOf(int64(0)),
            // Use a 0 value of type int64, check with gomock to ensure that the input must be of type int64
            gomock.AssignableToTypeOf(int64(0)),
            gomock.AssignableToTypeOf(&proto.ListAvailableTelevisionNetworksRequest{}),
        ).
        Return([]*bio.TelevisionNetwork{}, nil),
    Times(1)

    // ...
}

```

4. Common problems

During the above practice, we will encounter some common problems, and the corresponding solutions are presented below with specific cases.

(a) How to manage the generated mock file

In practice, we tend to treat the interface mock files as separate mock packages, generated and placed in the package where the corresponding interface is located. This way each package will be relatively independent and easy to manage. The following is a common structure of a microservice language project in the current team; you can see that the handler and model packages each contain the corresponding mocks package.

```

└── app.go
└── controller
    ├── handler
    │   ├── mocks
    │   │   └── handler1.go
    │   ├── handler1.go
    │   └── handler1_test.go
    └── model
        ├── mocks
        │   └── model1.go
        ├── model1.go
        └── model1_test.go
    └── regression
        └── regression1_test.go
    └── other_regressions_test.go
    └── scripts
        └── some_script_file
    └── config
        └── some_config
    └── proto
        └── some_proto
    └── go.mod
    └── go.sum
└── sonar-project.properties

```

(b) How many mock files need to be generated for an interface definition

The idea of incrementally adding mock files was mentioned above when describing gomock practices. If there is no test file that depends on an interface, then we do not recommend generating a mock file for it. This way, we can “generate when we need,” so that each code commit has a corresponding usage scenario.

Also, for an interface, we recommend using only one mock approach to generate its mock file. For example, if a microservice has adopted the testify/mock practice, it should try to continue this approach.

(c) Whether testing parallelism is required

A lot of articles on Golang unit tests mention optimizations through parallelism. However, we have found that in practice: the vast majority of unit tests themselves run very fast (usually in the millisecond range), which means that the scope for optimization through parallelism is very small and usually negligible; using `t.Parallel` imposes an extra mental burden for developers, such as the need to capture the variables of for ... range statement variables to circumvent the problem of test result coverage. Based on these two points, we do not recommend using `t.Parallel` in unit tests unless the corresponding tests are very time-consuming. The following is an example of a parallel run.

```
func TestGroupedParallel(t *testing.T) {
    for _, tc := range testCases {
        tc := tc //capture range variables
        t.Run(tc.Name, func(t *testing.T) {
            t.Parallel()
            if got := foo( tc.in); got != tc.out {
                t.Errorf("got %v; want %v", got, tc.out)
            }
            ...
        })
    }
}
```

(d) When to add a benchmark test

For most business code, there is little need to add benchmark tests, and we rarely add them in practice. There are currently only two exceptions.

- (i) Submit code to a base library or tool library, such as code that compresses the response to a request.
- (ii) The code to be tested has performance requirements; for example, a business goes live and finds by tracing that running a piece of code is very time consuming, and then it needs to be considered for performance evaluation through benchmarking.

8.2.2 Godog-Based Integration Testing Practices

Integration testing is performed after unit testing is complete, and it combines multiple units of code and all integrated services (such as databases, etc.) to test the correctness of the interfaces between them. As the core business architecture moves to a microservices architecture at a faster pace and as more services are built, we have designed different integration test cases for various services to minimize learning and testing costs when building new services.

Figure 8.4 depicts the integration testing process, which consists of four main phases: preparing test data, starting the test environment, running test cases, and generating test reports.

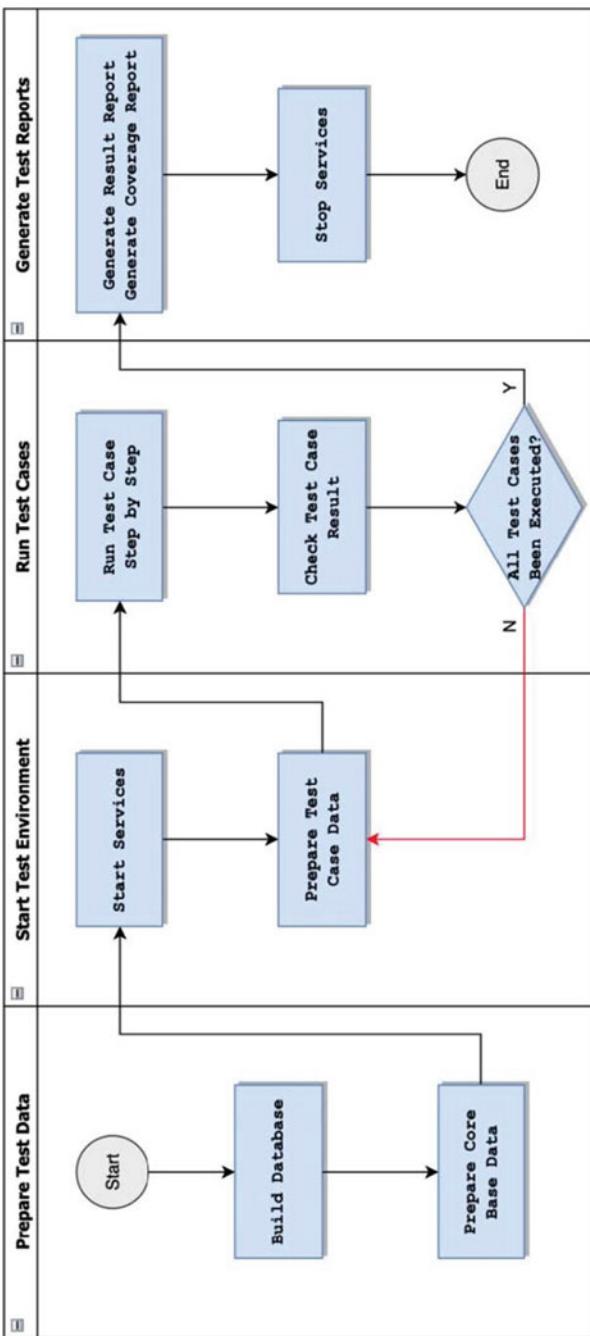


Fig. 8.4 Integration testing process

The above testing process is relatively clear and applicable to most teams, so we will not go into detail here. We have drawn a few lessons from our practice that are worth sharing with readers.

1. Test framework selection

Choosing a good testing framework can save effort and leads to better results, so you need to consider a variety of factors when selecting one, such as the following:

- (a) Whether it fits into the team's daily development and testing process
- (b) Whether it is easy to get started and maintain
- (c) Whether there is a high ROI

For our team, a natural idea was to select from the Golang language ecosystem.

There are three mainstream Golang testing frameworks: Ginkgo, GoConvey, and Godog. Of which Godog supports Gherkin syntax, which has two advantages.

- (a) As a Behavior-Driven Development (BDD) style framework, the writing syntax is close to natural language and easy to get started.
- (b) The team used the Cucumber testing framework during the monolithic application period, so the accumulated test cases were written based on Gherkin syntax, and the migration from Cucumber to *Godog* was very friendly to the team.

So, we ended up choosing to use Godog to write our test cases.

2. Test case preparation

In the phase of writing test cases, we follow the following steps:

- (a) Describe the test case behavior in the FEATURE file

Here is a template for a test case:

Feature: feature name

Background:

Given some preconditions
And some other state

Scenario: scenario name

When perform some action
And perform some other action
Then should make an assertion

- (b) Realization of the act

After defining the FEATURE file, it is then time to implement each step in the file scenario.

For example, for Given using the "Advertising" service, we can abstract the ^using the "(^"]*" service\$ generic step, where the content inside the quotes is a matching description of the variables in the step using a regular expression. In the actual runtime, Godog parses out the fields of the

Advertising and passes them into the appropriate step, executing the appropriate code logic via the `usingTheService` method.

So, we need to do two things.

First thing: register the relationship between step expressions and methods. The code is as follows:

```
func (a *APIFeature) RegisterSteps(s *godog.Suite) {
    s.Step(`^using the "([^\"]*)" service$`, a.usingTheService)

    // Other steps to register
}
```

where `APIFeature` is a predefined structure that we use to store some public data to be used between different steps and has the following structure:

```
type APIFeature struct {
    FeatureBase
    ServiceHost string
    RequestBody string
    StatusCode int
    ResponseBody []byte
    Schemas map[string]gojsonschema.JSONLoader
}
```

Second thing: Write code that implements the behavior expected by the steps:

```
func (a *APIFeature) usingTheService(service string) error {
    switch s := service; s {
    case "Advertising":
        a.ServiceHost = utils.Conf.AdvertisingService
    case "Other_Service":
        a.ServiceHost = utils.Conf.OtherService
    default:
        return fmt.Errorf("Undefined service: %s", service)
    }
    return nil
}
```

In addition to the framework and use cases, we need a primary database to provide the underlying data for the test runtime. Before all test cases are executed, the data tables needed for the tests are downloaded from the main database and saved as temporary SQL files. After each test case is completed, the temporary SQL file can be used to refresh the database and return it to its initial state, as shown in Fig. 8.5.

Using this strategy allows each test case to be tested based on the same database state, circumventing interference between test cases. At the same time, using the same public data source reduces the cost of data generation for each team and allows the data itself to be reviewed by multiple teams to ensure accuracy and timeliness.

However, in practice, we found that with the increase in test cases, the time spent on refreshing the database to the initial state takes up an increasing proportion of the overall integration test time. Therefore, the previous refreshing strategy needs to be optimized. Initially, we introduced a tagging mechanism to distinguish test cases that need to be refreshed. Here, we rely on Godog's own tag function, which characterizes the object being modified by

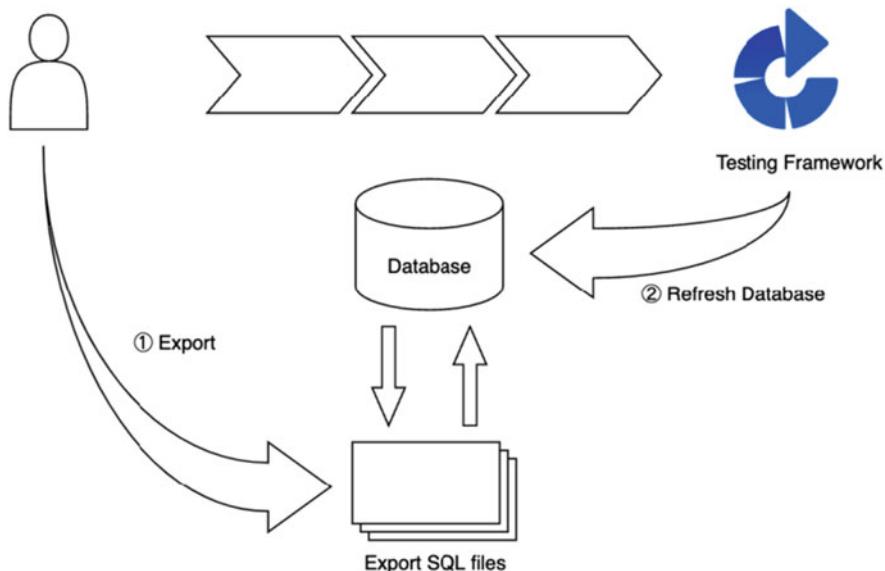


Fig. 8.5 Dump and refresh database during integration testing

specifying one or more words starting with @ on the Scenario, as shown in the following example:

```
@db_need_refresh
Scenario: create an advertiser
When ...
Then ...
```

However, there are two drawbacks to this solution, listed as follows:

- (i) These tags are just palliatives because read-only test cases are not a high percentage of the overall test set.
- (ii) The maintenance is difficult, and developers must remember not only to add the corresponding marker for a read-only test case but also to delete the marker in time when introducing write operations in that test case; otherwise, the test case will fail to run and may impact to other test cases which are hard to debug.

We later iterated on a new strategy, with the process shown in Fig. 8.6. For each test case, a corresponding rollback SQL statement is generated for each database write operation. When the test case is finished running, the data is rolled back using the rollback SQL file.

This strategy brings two benefits:

- (i) High performance because only rollback SQL statements need to be generated for the changed data, replacing global updates with local updates.
- (ii) Test case tagging is no longer required, as no rollback SQL statements will be generated for read-only test cases.

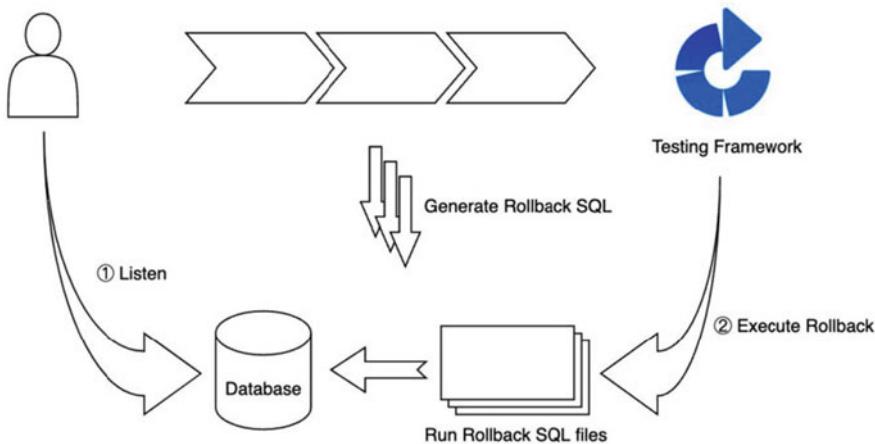
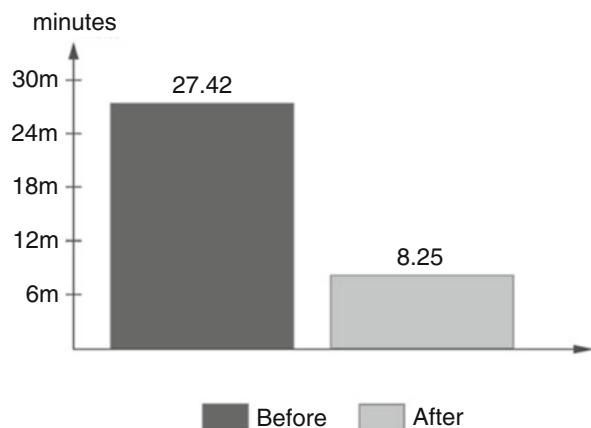


Fig. 8.6 Generate and Rollback SQL during integration testing

Fig. 8.7 Comparison of 2 database data refreshing strategies



After adopting this strategy, we selected a test set of a microservice for comparative testing. This test set contains 1772 scenarios with 16,111 steps. The test results are shown in Fig. 8.7. The comparison results show that the overall test time cost is reduced to about 30% of the original.

8.2.3 Cypress-Based End-to-End Testing Practices

End-to-end testing is testing from the user's point of view, where the software to be tested is considered as a black box, without needing to know the internal implementation details, and where the output is only concerned with expectations. In the following, we will present our practical experience in end-to-end testing:

1. Framework selection

In our team's microservice architecture lifecycle, the end-to-end testing segment has a broader applied range and treated at a higher place as the last line of defense to ensure the quality of the entire product line. In our previous monolithic architecture, we used a combination of Capybara, RSpec, and Selenium for end-to-end testing, but this testing approach gradually revealed many problems.

- (a) The test framework is complex to build and maintain and is not friendly to the CI pipeline.
- (b) The framework relies on the Ruby technology stack, contrary to Golang and JavaScript, which are the languages mainly used by the team.
- (c) Selenium Webdriver performance is so poor that the tests often fail.
- (d) Selenium has limited support for headless (headless) mode.

We did some exploring to get better at end-to-end testing in our current microservices architecture. Eventually Puppeteer and Cypress caught our attention. As shown in Table 8.1, we performed a multidimensional comparison of Puppeteer and Cypress.

An automated testing framework based on Puppeteer is an excellent solution, but, at the time of our exploring, the Puppeteer project was in its early stage, its core API was unstable, and the community ecosystem was in its infancy, so our team ultimately decided to use Cypress as the basis for an end-to-end testing framework.

2. Cypress practice

After the selection was finalized, our team implemented a Cypress-based end-to-end testing framework that can support automated testing of both Web UI and API.

(a) Support fixture

A fixture is an operation or script that creates the preconditions on which a test case depends during the software testing process, and these preconditions usually vary on the end-to-end test environment. There are at least three phases in our team's development testing process that require end-to-end testing.

- (i) Local testing: end-to-end local testing is required when code is in a custom branch that has not been merged into the main branch, and

Table 8.1 Puppeteer and Cypress comparison

	Puppeteer	Cypress
Installation difficulty	Low, just need to install Node.js	Low, just need to install Node.js
Degree of technology stack fit	Based on JavaScript, matching the front-end technology stack used by the team	Based on JavaScript, matching the front-end technology stack used by the team
Performance	Call the underlying browser wrapper API directly to take advantage of the performance benefits of V8	Specific optimizations for performance
Does it support failure test summation	The appropriate plugin needs to be installed to support	Comes with video recording and time travel features
Testing stability	High stability	High stability
Overhead	Open-source solution, relies on community ecology for development, many features need to be self-developed (Puppeteer was released shortly when the exploring was done, and the community was still in the early stages of construction)	Core code is open source and provides a commercial solution
Browser compatibility	Chrome, Firefox (Puppeteer was released shortly when the exploring was done and only supported Chrome)	Chrome, Firefox (Cypress only supported Chrome at the time of exploring)

developers complete functional testing by adding new end-to-end test cases.

- (ii) Regression testing: end-to-end regression testing is required after the functional code is merged into the main branch.
- (iii) Post-release check testing: After a feature is released to the online environment, end-to-end testing is required to implement post-release checks to ensure that the feature still works as expected in the online environment.

Based on the above, in order to maximize the reusability of end-to-end test cases and to consider the complexity of building a local end-to-end test environment, we have added support for fixture to the testing framework. For example, suppose there is an existing test scenario where the status of a particular order is checked, the order number may be different between the online and development environments, and some other information related to the order may be different in addition to the order number. In this case, a fixture can be used, with the following code:

```
// fixture is used to indicate the environment in which to execute the test case
const fixture = {
  prd: {
    networkInfo: Cypress.env('prdTestNetWorkInfo'),
    orderId: 26341381
  },
  stg: {
    networkInfo: Cypress.env('stgTestNetWorkInfo'),
    orderId: 26341381
  },
  dev: {
    networkInfo: Cypress.env('localNetWorkInfo'),
    orderId: 133469319
  }
};
const { networkInfo, orderId } = fixture[Cypress.env('TEST_ENV')];
```

(b) tag capability

After applying fixture to each test flow, it is also important to consider a scenario where the test cases that need to be run in different environments may be different. For post-release check tests in an online environment, the highest level of test cases needs to be run, which are small in number but cover the core functionality of the entire system. For day-to-day end-to-end regression testing, a larger range of test cases needs to be run. To meet this requirement, the team added tagging functionality to the framework to categorize the test cases.

```
// tag is used to indicate that this is a P1 level test case
Cypress.withTags(describe, ['p1'])(
  'Create order', function() {
    // 测试 case 代码
  }
);
```

(c) Test cases

A typical Cypress test case is as follows, requiring the provision of before and after hooks for data initialization and data scavenging, respectively, and then writing the core test logic in the statement.

```
Cypress.withTags(describe, ['p1'])(
  'Create IO', function() {
    before(function() {
      cy
        .loginByUI(networkInfo, '/campaigns/${campaignId}/edit')
        .waitUntilLoaded();
    });

    it('Create an empty order', function() {
      cy.createEmptyOrder()
        .get('#order_id').eq(0)
        .invoke('text')
        .then(order_id => {
          orderID = order_id;
        });
    });

    after(function() {
      cy.deleteOrder({ orderID });
    });
  });
};
```

(d) Headless mode

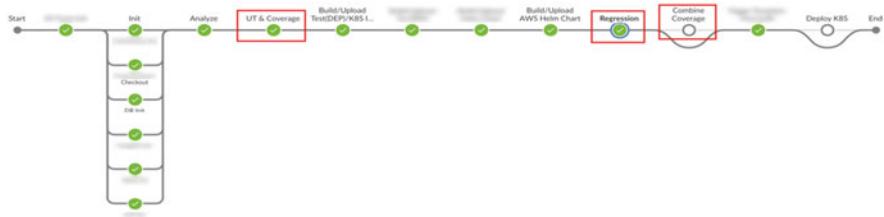


Fig. 8.8 Continuous integration pipeline

Cypress comes with support for headless mode. For example, if you execute the cypress run command, it will launch the Electron browser by default and run the test in headless mode. For example, a common configuration is Cypress open --browser chrome.

End-to-end testing ensures that the tested product meets the user's needs, but its test granularity is too large and requires a stable of the test environment and high-quality test data, resulting in high maintenance costs. Thus, every time you write an end-to-end test case, you must think about whether it is necessary.

8.2.4 *Test Automation*

To improve development efficiency, detect problems early, reduce repetitive work, and automate testing, we have integrated the Jenkins pipeline to implement continuous integration and continuous deployment. The specific practices of CI and CD are detailed described in Chap. 9 and are only briefly described here.

1. Continuous integration phase

There are two general trigger points for a continuous integration pipeline:

- Continuous integration tests are triggered before the code is merged into the main branch, and the code is allowed to be merged into the main branch only after the tests pass.
- Code merging into the main branch triggers a continuous integration test, which is designed to verify that the main branch meets quality expectations.

Figure 8.8 shows the flow by the blue ocean plugin of the Jenkins pipeline defined by the Groovy script, and several important phases related to testing will be analyzed in the following with specific cases, including the unit testing and coverage reporting phase (UT & Coverage), the integration testing and coverage reporting phase (Regression, Combine Coverage), and the code coverage notification phase.

- Unit testing and coverage reporting phase

Simply specify the script used by the unit test in the Groovy script and turn the Generate Coverage switch on to get the test coverage report as follows:

```
stage('UT & Coverage'){
    ... // some code...
    environment {
        core_common = get_core_common(serviceFullName)
        // get UT test coverage report
        ut_cobertura_report_file = get_ut_cobertura_report_file(serviceFullName)
    }
    steps {
        //specify shell script to execute ut cases
        sh(returnStdout: true, script: "sh ${WORKSPACE}/shell_scripts/unit_coverage.sh")
    }
    post {
        success {
            // if successful, generate a UT test coverage report in HTML format
            archiveArtifacts allowEmptyArchive: true, artifacts: ut_cobertura_report_file, fingerprint: true
            sh 'echo "ci.ut.result=PASS" >> ${WORKSPACE}/env.props'
        }
    }
}
```

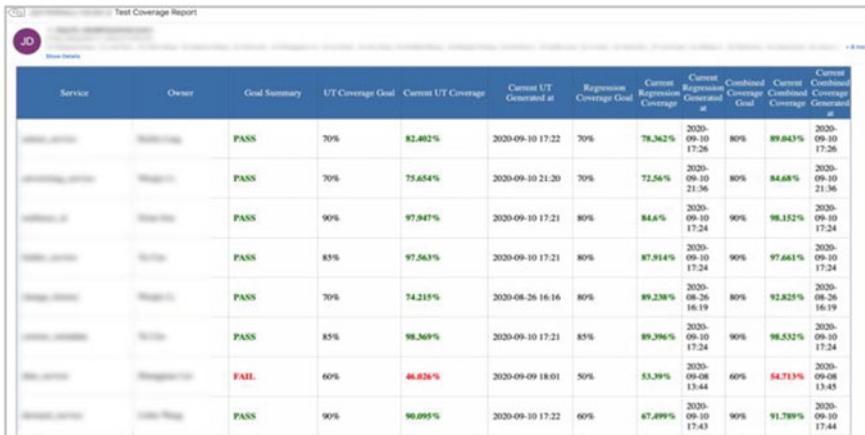
(b) Integration testing and coverage reporting phase

The integration test coverage report is obtained in a similar manner to the unit tests, by simply specifying the script used for regression testing in the Groovy script, and the continuous integration pipeline will output the generated results as follows:

```
stage("Regression"){
    environment {
        ...// some code...
        html_report_dir = get_report_dir(serviceFullName)
        // Get Regression test coverage report
        regression_cobertura_report_file = get_regression_cobertura_report_file(serviceFullName)
        diff_files ="${WORKSPACE}/diff/*"
    }
    steps {
        sh """
            mysql -uroot -proot -h127.0.0.1 -e "source ${WORKSPACE}/sql/ui_permission_sql.sql"
            // Regression test data preparation
            ${WORKSPACE}/Integration_test_data/bin/initDB.sh
            // Regression test environment preparation
            ${WORKSPACE}/shell_scripts/regression_init.sh
            if [[ ! -d ${html_report_dir} ]]; then
                mkdir ${html_report_dir}
            fi
            ""
            // Specify the execution script for the Regression test case
            sh "${WORKSPACE}/regression_scripts/${serviceFullName}_regression.sh"
    }
    post {
        success {
            ... // some code...
        }
    }
}
```

(c) Code coverage notification phase

FreeWheel's products are mainly designed for enterprise-level customers, and a code coverage target of 90% has been set at the company level for ensuring product quality. This value is the result of combining the coverage rate of unit and integration tests. To achieve this goal, we took a step-by-step approach to increase coverage, setting separate timelines for unit and integration tests to meet the target. During the development cycle of each new product release, a Groovy script is used to set the test coverage target to be



The screenshot shows a 'Test Coverage Report' email with a table of coverage data. The columns include Service, Owner, Goal Summary, UT Coverage Goal, Current UT Coverage, Current UT Generated at, Regression Coverage Goal, Current Regression Coverage, Current Regression Generated at, Combined Coverage Goal, Current Combined Coverage, and Current Combined Coverage Generated at. The data is as follows:

Service	Owner	Goal Summary	UT Coverage Goal	Current UT Coverage	Current UT Generated at	Regression Coverage Goal	Current Regression Coverage	Current Regression Generated at	Combined Coverage Goal	Current Combined Coverage	Current Combined Coverage Generated at
...	...	PASS	70%	82.402%	2020-09-10 17:22	70%	78.362%	2020-09-10 17:26	80%	89.845%	2020-09-10 17:26
...	...	PASS	70%	75.654%	2020-09-10 21:20	70%	72.56%	2020-09-10 21:36	80%	84.68%	2020-09-10 21:36
...	...	PASS	90%	97.947%	2020-09-10 17:21	80%	84.6%	2020-09-10 17:24	90%	98.152%	2020-09-10 17:24
...	...	PASS	85%	97.563%	2020-09-10 17:21	80%	87.914%	2020-09-10 17:24	90%	97.661%	2020-09-10 17:24
...	...	PASS	70%	74.215%	2020-08-26 16:16	80%	89.238%	2020-08-26 16:19	80%	92.825%	2020-08-26 16:19
...	...	PASS	85%	98.369%	2020-09-10 17:21	85%	89.396%	2020-09-10 17:24	90%	98.532%	2020-09-10 17:24
...	...	FAIL	60%	46.826%	2020-09-09 18:01	50%	53.39%	2020-09-08 13:44	60%	54.713%	2020-09-08 13:43
...	...	PASS	90%	90.095%	2020-09-10 17:22	60%	67.499%	2020-09-10 17:43	90%	91.789%	2020-09-10 17:44

Fig. 8.9 Test coverage report email example of unit test and integration test

achieved for that release. Any code that fails a test or does not meet the coverage target will not pass the merge request and will be notified via Slack.

```
stage('Coverage & Analyze'){
    ...//some code...
    post {
        success {
            // determine if test coverage targets are met
            cobertura autoUpdateHealth: false, autoUpdateStability: false, coberturaReportFile: combined_cobertura_report_file,
            conditionalCoverageTargets: '70, 0, 0', failUnhealthy: false, failUnstable: false, lineCoverageTargets: '80, 0, 0', maxNumberOfBuilds: 0,
            methodCoverageTargets: '80, 0, 0', onlyStable: false, sourceEncoding: 'ASCII', zoomCoverageChart: false
            archiveArtifacts allowEmptyArchive: true, artifacts: "${html_report_dir}/*.json", fingerprint: true
        }
    }
    post {
        failure {
            // If not reached, send a message via Slack to notify the appropriate people
            slackSend channel: "#${slack_channel}", color: "danger", message: "AWS Build FAIL! ${serviceFullName}"
            <${BUILD_URL}|${BUILD_DISPLAY_NAME}> ${currentBuild.description}"
        }
    }
}
```

As shown in Fig. 8.9, each week we also collect the test coverage results of unit tests and integration tests and send notifications via email, urging relevant microservice teams to make up the corresponding tests in time to bring the coverage rate up to the standard.

2. Ongoing deployment phase

Once the product has been deployed online, a Jenkins task for end-to-end testing can be triggered via a pipeline correlation trigger to perform testing related to the product once it goes live. As mentioned earlier, our end-to-end testing uses Cypress as the foundation of the framework, which supports integration with Jenkins. In our continuous deployment script, we can set up the test results to be sent out via both email and Slack, which greatly reduces the response time for faulty test cases and improves product quality. The following script code shows

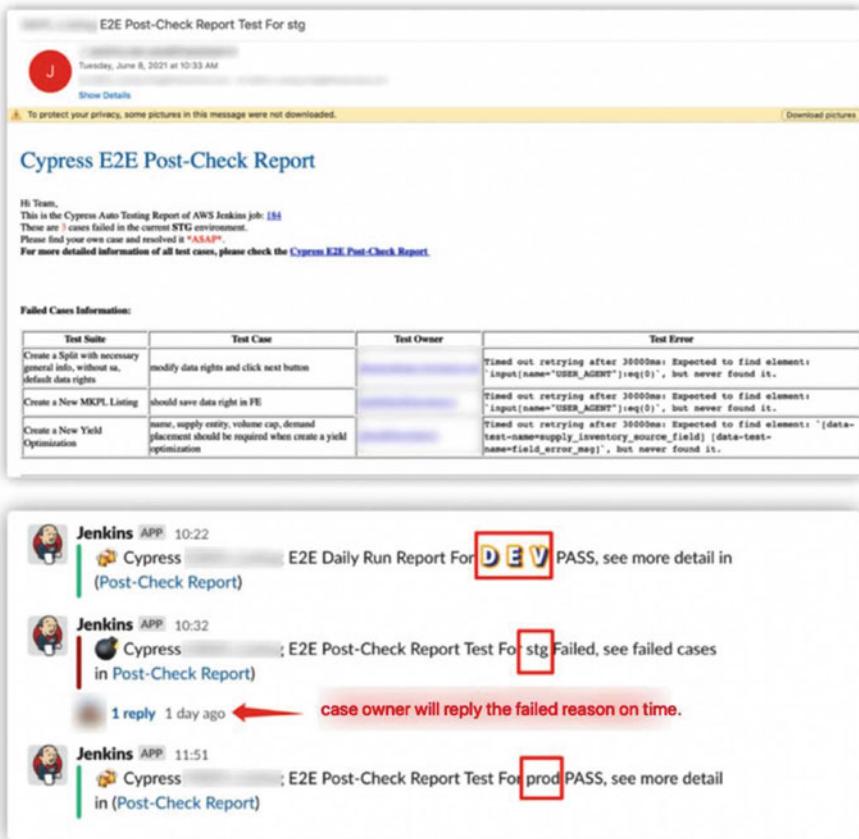


Fig. 8.10 A E2E post-check report email example and corresponding slack message example

the logic for handling tests after they succeed and fail. Figure 8.10 shows the results of the notifications received via email and Slack.

```

pipeline {
    post {
        success {
            publishReport()
            notifySuccess()
            sendMail()
        }
        // If the end-to-end test fails, send an email and Slack message to notify the appropriate people
        failure {
            publishReport()
            archiveArtifacts artifacts: 'screenshots/**, videos/**'
            notifyFail()
            sendMail()
        }
    }
}

```

8.3 Chaos Engineering

Distributed systems like microservices are always faced with stability issues. While monitoring and alerting can help with troubleshooting, these are all remedial actions after a failure has occurred. As the saying goes, it is better to prevent fires than to put out them, and it is crucial to know in advance when an application will go wrong. Chaos engineering is the discipline of experimenting on distributed systems in order to build the capability and confidence of the system to withstand runaway situations in a production environment. This section will introduce the core concepts of chaos engineering and introduce our team's practice in this area.

8.3.1 *The Core Concept of Chaos Engineering*

Netflix's Chaos Engineering team defines chaos engineering on the "Principles of Chaos" website (principlesofchaos.org) as follows:

Chaos engineering is a discipline that conducts experiments on systems to build the capability and confidence of the system to withstand runaway situations in production environments.

The goal of chaos engineering is doing experiments on a system, finding out what flaws the system has based on the experiment results, and then finding countermeasures to avoid failures or minimize the damage to the system when they occur. By continuously conducting iterative experiments, we can build up a stable and reliable system, which will not only serve our customers well but also reduce the number of engineers being called up at the midnight to deal with failures.

Chaos engineering experiments (hereafter referred to as chaos experiments) are very different from traditional tests. Traditional tests will define the inputs and outputs in advance, and if the result doesn't meet the expectation, the test case will not pass. Chaos experiments are not like this; the result of the experiment is uncertain. For example, network latency, CPU overload, memory overload, I/O anomalies, etc., we didn't know what kind of impact they have to the systems. Some of them we can predict, and some of them we cannot know until they happen. Ideally, chaos experiments become part of the team's daily work; each member can take on the role of chaos engineer and can have the opportunity to learn from the experiment results.

We will further introduce chaos engineering in terms of its development history, principles, and maturity models below.

1. History of chaos engineering

Chaos engineering has been evolving for over 10 years. Some of the popular implementations of chaos engineering in the industry include Netflix, Alibaba, PingCAP, and Gremlin. Netflix is the most representative as they have built some form of resilience testing since they moved their data center to the cloud in 2008, which they later called chaos engineering. Since then, they have continued to

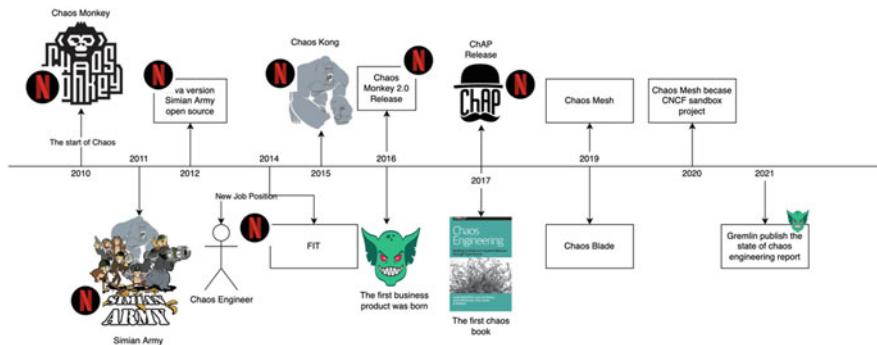


Fig. 8.11 Chaos engineering development process

explore and evolve chaos engineering, and they have contributed a lot to the development of chaos engineering. The timeline of the development of chaos engineering is shown in Fig. 8.11.

- In 2010, Netflix developed the Chaos Monkey, a tool that randomly disables AWS EC2 instances. The Chaos Monkey project was the beginning of Chaos Project.
- In 2011, Netflix came up with the Simian Army tool set. In addition to Chaos Monkey, the toolset includes seven other types of monkeys, such as Latency Monkey, which can introduce artificial delays in RESTful requests to simulate service degradation and downtime by adjusting the length of the delay, and Janitor Monkey, which can be used to monitor unneeded resources and recycle them. In 2012, Netflix open-sourced a Java version of the Monkey Army toolset to the community, including Chaos Monkey 1.0, which is no longer maintained now.
- In 2014, Netflix created the position of Chaos Engineer. In October of the same year, Netflix proposed Fault Injection Testing (FIT), a tool based on Monkey Army that allows developers to control the scope of chaos experiments from a finer granularity.
- In 2015, Netflix released Chaos Kong, which can simulate failures in AWS regions, and formally introduced the principles of chaos engineering in the community—the Chaos Engineering Principles. These principles became the guidelines in doing chaos experiments.
- In 2016, Gremlin was founded, and chaos engineering started to have a commercial product. The founder of Gremlin is a former Chaos engineer at Netflix. That same year Netflix released Chaos Monkey 2.0.
- In 2017, Netflix released the Chaos Experiment Automation Platform (ChAP), which can be considered an enhanced version of the fault injection testing tool, with improved security, pacing, and scope control of experiments. The first book introducing the theory of chaos engineering was also released this year.

- (g) In 2019, PingCAP released Chaos Mesh, a cloud-native open source project, and Alibaba also open-sourced Chaos Blade, a chaos engineering tool based on nearly a decade of Alibaba's fault testing practices.
- (h) In July 2020, the Chaos Mesh platform became a sandbox project of CNCF.
- (i) In early 2021, Gremlin released its first State of the Chaos Engineering Industry report.

We believe that with the development of Cloud Native, domestic companies will pay more and more attention to chaos engineering, and more people will recognize the concept of chaos engineering, and chaos engineering will be further developed.

2. Principles of chaos engineering

The word chaos is often used to describe confusion, but chaos engineering is not chaotic; on the contrary, it has a well-established guiding philosophy. As we can see from the definition, chaos engineering is a discipline about experimentation, and there is a theoretical system which is recognized as the "Chaos Engineering Principles," which are the lessons learned and best practices of Netflix in implementing chaos engineering.

There are five major principles to follow when running chaos experiments: build a hypothesis around steady-state behavior, vary real-world events, run experiments in a production environment, automate experiments to run continuously, and minimize the blast radius. We describe each of them below:

(a) Build a hypothesis around steady-state behavior

Steady state refers to the state of a system when it operates normally, such as the utilization rate of system resources being within the set range, customer traffic remaining stable, etc. This state is referred to as the steady-state. It is just like our body temperature; above 37 °C we will feel feverish and need to rest and drink a lot of water to bring our body temperature back to normal, while once our body temperature is above 38.5 °C, we may need to use drugs or other treatments to help our body return to normal. It is similar to defining a steady-state for a system, which needs to rely on data indicators that can be quantified.

Usually, we divide the metrics into system metrics and business metrics. System metrics can be used to characterize the health of the system, and business metrics can be used to indicate whether the customer is satisfied with the system.

It is relatively easy to obtain system metrics; you can get CPU load, memory usage ratio, network throughput, disk usage, and other metrics through the monitoring system and analyze these data and set a threshold value; you can use this threshold value as a steady-state system indicator; once the indicator is not within the threshold value, it can be taken as the steady-state has been damaged, and the steady-state damage often causes the monitoring system alarm.

In contrast to system metrics, business metrics are not so easy to obtain and have to be analyzed case-by-case. Take order conversion as an example;

based on historical data, we can get a rough order conversion rate, or a periodically changing order conversion rate, plus a forecast of the future market; we can define a steady-state conversion rate for the function of generating orders, and if the order conversion rate suddenly appears too high or too low, we can consider the steady-state to be disrupted. However, we cannot just assume that the steady-state disruption is a bad thing; it is also an opportunity for us to learn the behavior of the system. We have found in practice that business metrics often need to be obtained by some other technical solution.

(b) Vary real-world events

Real-world events are a wide variety of problems that a system may encounter during operation, such as hardware failures, network delays, resource exhaustion, sudden service downtime, etc. When running chaos experiments, it is impossible to simulate all failures because of too many events. It is necessary to consider the frequency and impact scope of these failures and then choose those which will occur frequently and have big impact. In our practice, we prefer the events that have caused failures and bugs many times before. For example, network latency is one of the most likely events to cause problems in the system.

(c) Run in production environment

For this principle, people may mistakenly think that chaos experiments can only be run in a production environment. In reality, this is not the case, and it would be irresponsible to run chaos experiments directly in a production environment. It is better to run chaos experiments in a test environment at the beginning, and then we can run experiments in a production environment until we have enough confidence.

In early 2021, Gremlin counted in the State of the Chaos Engineering Industry report that only 34% of participants expressed confidence in running chaos experiments in a production environment. It shows that running chaos experiments in a production environment is not very mature now.

The reason of chaos engineering in proposing “running in a production environment” as an important principle is that the test environment is often very different from the production environment, and it is difficult to simulate the production environment in terms of resource allocation or business data. The goal of chaos engineering is to ensure that the business in the production environment is continuous and uninterrupted, so running chaos experiments in the production environment makes it easier for us to build resilient systems. With the development of cloud-native technologies, more and more systems are built on cloud platforms, and it becomes quick and easy to build production-like environments, which allows us to run chaos experiments more frequently to gain confidence and accelerate our pace of running chaos experiments in production environments.

(d) Automate experiments to run continuously

Manually run experiments are unreliable and unsustainable, so chaos experiments must be implemented automatedly. Continuous run chaos experiments can ensure business continuity during the system iterations.

Automation includes three aspects: running experiments automatedly, analyzing the experimental results automatedly, and creating experiments automatedly.

Running experiments automatedly can be achieved through automated platforms (e.g., Netflix's ChAP platform) or with the help of some mature tools (e.g., Jenkins), such as our practice of integrating chaos experiments into CI/CD pipelines and running chaos experiments regularly.

Analysis of the experiment results automatedly includes automatedly monitoring of the system data, terminating the experiment if anomalies are detected, and raising problems that can cause a system failure. The ability to automate the operation of the experiment and to automate the analysis of the experiment results is considered as an excellent practical solution for chaos experiments.

Creating experiments automatedly is an advanced stage of chaos experiments. It is worth mentioning that the lineage-driven fault injection (LDFI) technique has been proposed, which can be applied to automate the process of creating experiments. This technique was developed by Professor Peter Alvaro of the University of California, Santa Cruz. At QCon London 2016, Peter Alvaro shared a successful collaboration with Netflix using this technique, which found a new path for Netflix to automate fault injection testing and provided a direction for us to study automated experiment creation.

(e) Minimize blast radius

Chaos experiments are run to explore unknown problems that can cause system failures. In searching for these unknown problems, we want to be able to expose them without causing larger failures due to unexpected events. It is known as “minimizing the blast radius.”

The reason is that there are risks associated with chaos engineering; the biggest risk is the possibility of crashing the system and impacting the customers, which is the resistance that can be encountered in running chaos experiments. A step-by-step approach is used for each experiment to control the risks associated with chaos experiments in terms of both the scale of the experiment and the impact on the customers.

For example, start with a minimal scale experiment, targeting only a very small number of users or traffic and injecting faults into only a small portion of the service. Once the small-scale experiments are successful, the next step is to spread the experiments, still choosing to inject faults into small flows, and follow the normal routing rules so that the experiments are evenly distributed in the production environment. Further centralized experiments can be performed, such as setting up faults, delays on specific nodes, which ensures that the impact on customers is manageable. To reduce the impact on customers, we should try to run chaos experiments at times when the traffic is low, and engineers are in the office. Chaos engineers have the responsibility

and obligation to ensure that risks are minimized and to be able to stop the experiment at any time if it causes too big harm.

The above principles have been summarized by Netflix engineers through practice and are very helpful to guide us in running chaos experiments.

3. Chaos Engineering Maturity Model

As we all know, Capability Maturity Model (CMM) is a metric system for software maturity, which is used to define and describe what kind of capabilities software should have in each stage of development practice. Chaos engineering has also developed a theoretical model that can be used to measure the implementation of chaos engineering and guide the direction of chaos experiments. This model is the Chaos Maturity Model mentioned in the book “Chaos Engineering: The Way to Netflix System Stability.” Netflix has also divided it into two dimensions.

(a) Proficiency level

Proficiency is used to reflect the effectiveness and safety of experiments. Based on practical experience, Netflix classifies proficiency into four levels, the higher levels indicating the high validity and safety of chaos experiments. Table 8.2 describes these four levels from different perspectives.

(b) Acceptance level

The Chaos Experiment’s acceptance level reflects the team’s knowledge of chaos engineering and confidence in the system. Like proficiency levels, Netflix also divided the acceptance into four levels. Higher levels indicate the team’s deeper knowledge of chaos engineering and confidence in the system, as shown in Table 8.3.

Combining the proficiency level and acceptance level of the chaos engineering experiments, Netflix has plotted the Chaos Engineering Maturity Model as a map with proficiency as the vertical coordinate and acceptance as the horizontal coordinate, and the plot is divided into four quadrants as shown in Fig. 8.12.

The team can easily locate the current status of chaos engineering on the map and see its future direction. For example, a team has developed a tool which can automate the running of chaos experiments; they can place this tool at the right place on the map. From the map, they can find the tool itself is easy to use, but it needs to be improved in terms of acceptance in later work. Chaos engineers can focus their efforts on the promotion of this automation tool.

As technology develops and the understanding of chaos engineering increases, more and more teams will perform chaos experiments on their systems. The Chaos Engineering Maturity Model indicates the current direction of chaos experiments and can help you understand what a more mature and stable practice is.

Table 8.2 Chaos proficiency level

Proficiency level	Level 1 (elementary)	Level 2 (simple)	Level 3 (skilled)	Level 4 (advanced)
Description	Commonly found in organizations that are new to chaos engineering and are in their infancy	It basically reflects that the organization has developed or used some tools to assist in the completion of chaos experiments	Reaching this level indicates that the organization has the confidence to do experiments in a production environment	In the current state of development of chaos engineering, this level is the highest level for running chaos experiments
Experimental environment	Only dare to run experiments in development and test environments	Run experiments in an environment that is highly similar to the production environment	Run experiments in production environment	Experiments can be run in any environment
Experimental automation capabilities	All manual operation	Run experiments automatically, but need to manually check monitor and manually terminate experiments	Run and terminate experiments and monitoring automatically, but the analysis of results requires human intervention	All automated, including terminate experiments when abnormal cases occur
Event selection	Injecting simple faults, such as killing processes, disconnecting the network, etc.	Inject higher-level events such as network latency, network packet loss	Injecting combined failures	Injected events are more complex and detailed, such as changing the state of the system or changing the returned results, etc.
Experimental measurement design	Experimental results reflect only the system state	Experimental results can reflect the health of individual applications	Experimental results reflect aggregated business metrics	Comparisons can be made between experimental and historical results
Experimental tools	No tools	Use of experimental tools	Experimental framework and continuously develop tools	Experimental tools support interactive comparisons
Experimental results processing capability	Manual collation, analysis and interpretation of experimental results	Experimental tools allow for continuous collection of experimental results but still require manual analysis and interpretation of results	Experimental tools for continuous collection of experimental results and can do simple failure analysis	Based on the experimental results, the experimental tool not only allows for failure analysis but also predicts revenue loss, performs capacity planning, and

(continued)

Table 8.2 (continued)

Proficiency level	Level 1 (elementary)	Level 2 (simple)	Level 3 (skilled)	Level 4 (advanced)
				distinguishes the actual criticality of different services
Confidence in the system	No confidence	Confidence in some features	Have confidence	Resilient system with confidence

Table 8.3 Acceptance level

Acceptance level	Level 1 (in the shadows)	Level 2 (officially approved)	Level 3 (set up a team)	Level 4 (becoming cultural)
Experimental legality	Experiments were not approved	The experiment was officially approved	Officials set up a special team for chaos experiments	Chaos experiments become part of the engineer's daily work
Experimental scope	Few systems	Multiple systems	Most core systems	All core systems
Experimental participants	Individuals or several pioneers	Some engineers, from multiple teams, work part-time on chaos experiments	Professional team	All engineers
Experimental frequency	Try occasionally	Unscheduled runs	Regular runs	Normalization

8.3.2 How to Run Chaos Experiments

Running chaos experiments is to discover the unknown issues in the system, not to verify the known issues. Please ensure that there are no critical known issues in the system before running chaos experiments. From the communications and discussions in the chaos engineering community, it is not only large Internet companies that can run chaos experiments, but any team can choose a suitable development path to iteratively run chaos experiments. The chaos experiment itself is a closed cycle system of continuous iteration, which includes seven steps. Figure 8.13 shows the steps of running a chaos experiment.

1. Design experimental scenarios

Before starting an experiment, the first thing to clarify is what the experiment is going to be. We usually design experimental scenarios by asking questions such as: If the main instance of Redis goes down, will the entire system become unavailable? What happens to the web page the user is viewing when the process is suddenly killed? Let's use a real-world application scenario as an example. We have a forecasting service, and the customer can trigger the forecasting function

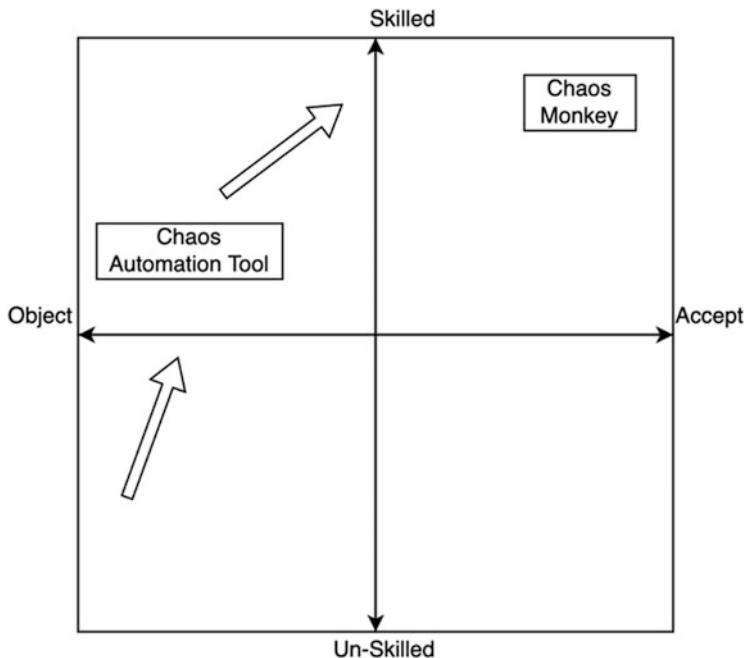


Fig. 8.12 Chaos engineering maturity model

by calling one of the APIs and can also get the forecasting data by calling another API. Based on the above, we choose the experimental scenario shown in Table 8.4.

Different business scenarios or architectural designs will generate different experimental scenarios. Table 8.5 lists some common scenarios used in chaos experiments, which readers can refer to when designing experiments.

2. Select experimental scope

Once the experimental scenario is defined, the next step is to determine the scope of the experiment. The selection of the experimental scope should follow the principles “running in a production environment” and “minimizing the blast radius.” Under the immature chaos experiments condition, I suggest you should not try it in a production environment; choose a suitable place for yourself and migrate it to the production environment step by step. You can simulate the production environment and make it as similar as possible to the production environment. This can ensure the effectiveness of the experiment.

Moving on to the scenario mentioned in Step 1, we describe the scope of the experiment like this. We built an experimental environment according to Fig. 8.14, in which the data and requests are all from the production environment. By analyzing the business code, we choose the API for obtaining prediction data which only reads data from the service; even if an exception occurs during the

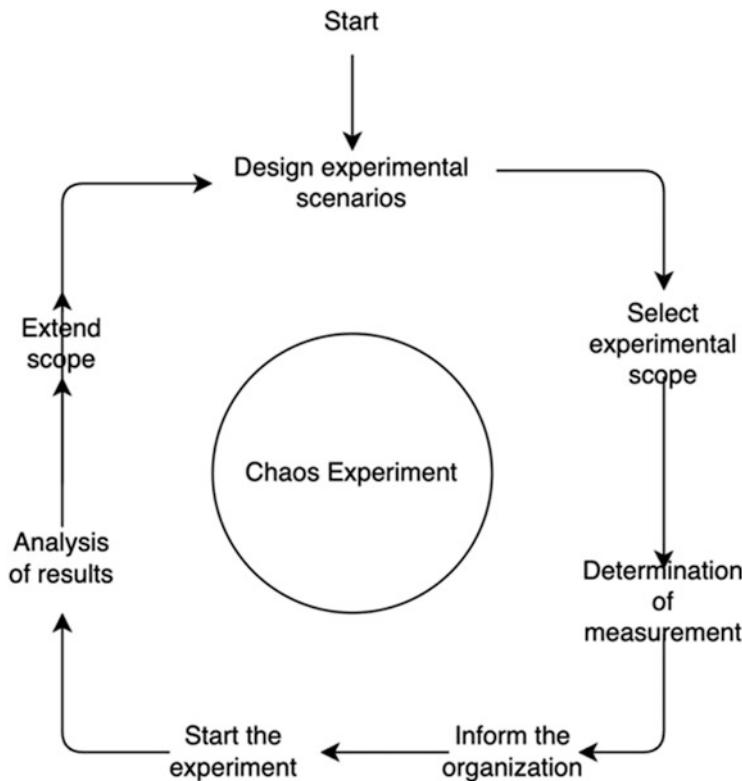


Fig. 8.13 Chaos experiment steps

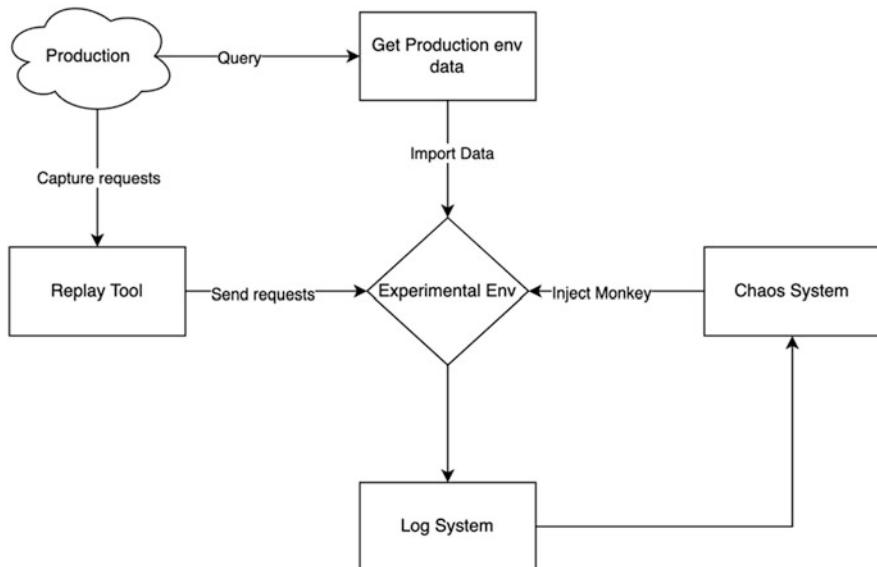
experiment, it does not generate dirty data and nearly no impact on the whole system. By observing the online logs, we also found that clients have a retry mechanism when using this API, which also boosted our confidence in running chaos experiments. We hope that readers can make a reasonable choice when choosing the scope of experiments by analyzing business implementation and user usage experiences.

Table 8.4 Experimental scenarios

Scene description	What will happen
CPU usage reaches 100% for all Pods running the service	Whether customers can still use the service normally and whether they need to restrict the flow of customers
For All Pods running this service, CPU usage is restored from 100% to normal	Whether the service's capacity can also be restored to normal, and how long it will take to do so
A portion of the Pods running this service has higher CPU usage	Whether the service will be automatically expanded

Table 8.5 Common scenarios

Experimental scenes	Example
Application layer failures	Abnormal process deaths, traffic spikes
Dependency failure	Dependent service down, database master instance dead
Network layer failure	Network latency, network packet loss, DNS resolution failure
Base resource failure	CPU overload, memory exhaustion, disk failure, I/O abnormalities

**Fig. 8.14** Chaos experiment environment architecture**Table 8.6** Steady-state metrics

	Peak value	Average	Stable
Volume of requests	30TPS	20TPS	>30TPS
CPU	23%	20%	≤100%
Response time	<3 s	<3 s	<3 s

3. Determination of measurement

By determining the experimental scenario and the experiment scope, we can get the metrics used to evaluate the experimental results. These metrics are always used to define the steady state of the system as mentioned in the principles of chaos engineering.

Referring to the forecasting service in Step 1, we promise the customer that the return time of each request for this service would not exceed 3 s. By analyzing the customer's access and system metrics, we defined a steady-state metric from a business perspective, as shown in Table 8.6.

When the traffic is too high, requests that couldn't return in 3s will be considered as breaking the steady state; this will affect the customer's business. After determining these metrics, we also needed a clear definition of "unexpected"; if the impact of the experiment was more severe than expected, the experiment had to be stopped immediately. For this experiment, we specify that once we find that 5% of the customer's traffic cannot respond within 3 s, we need to stop this experiment immediately and restore the system resources.

4. Inform the organization

If you are running chaos experiments in a production environment, this step will be very important at the beginning. You must let everyone know what you are doing, why you are doing it, and what is likely to happen. This process may require a lot of face-to-face communication to make sure that the members of the organization understand what the chaos experiment is trying to do. When the experiment becomes the norm, everyone will get used to it and the cost of communication will come down. At that point, it might be enough to send an email to notify team members. Eventually, the ideal situation is that running chaos experiments becomes part of the team's daily routine work, and it is no longer needed to notify organization members before each experiment.

5. Start the experiment

If running experiments in a pre-release environment, Chaos engineers can decide when to run the experiments themselves. If running experiments in a production environment, it is best to choose office hours so that the relevant people can be found to deal with them if something unexpected happens. There are many tools available to support your experiments. You can use open source tools such as Chaos Mesh and commercial software such as the Chaos Engineering Platform developed by Gremlin, or you can build your own tools for your business. If you do not intend to invest too much in chaos engineering, just want to try a simple experiment, using the kill command is a good choice.

We have developed a set of cloud-native chaos experiment tools. The experimental scenario mentioned in Step 1 uses the BurnCPU function in our tool, and the implementation principle will be described in detail in the later sections.

6. Analysis of results

After the experiments finished, the results need to be analyzed. Most of the problems exposed by chaos experiments are not a single service problem and need interactions between services, so the results need to be fed back to the development team in time. The results of the experiments will reflect not only the defects of the system but also the problems of the experiments themselves, which will help the chaos engineering team to make further improvements to the chaos system and enhance the steps of conducting the experiments.

In our experiments, the logs are collected through the company's internal monitoring system. Figures 8.15 and 8.16 show the results we collected. The beginning and end part of each of these graphs are the data without running BurnCPU, and the middle part of the response time increase is the result after running BurnCPU. Figure 8.15 shows the response time when the system throughput TPS is 20 requests/s and CPU occupancy is 100%. Figure 8.16

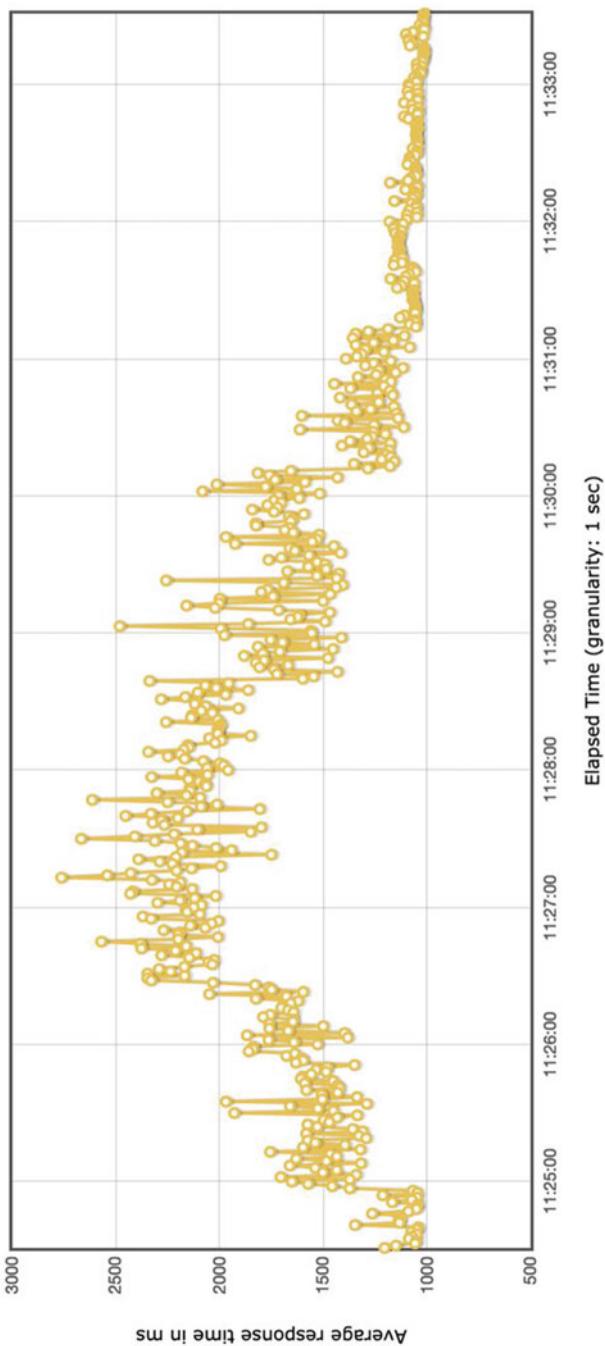


Fig. 8.15 Response time when TPS=20/s in burn CPU experiment

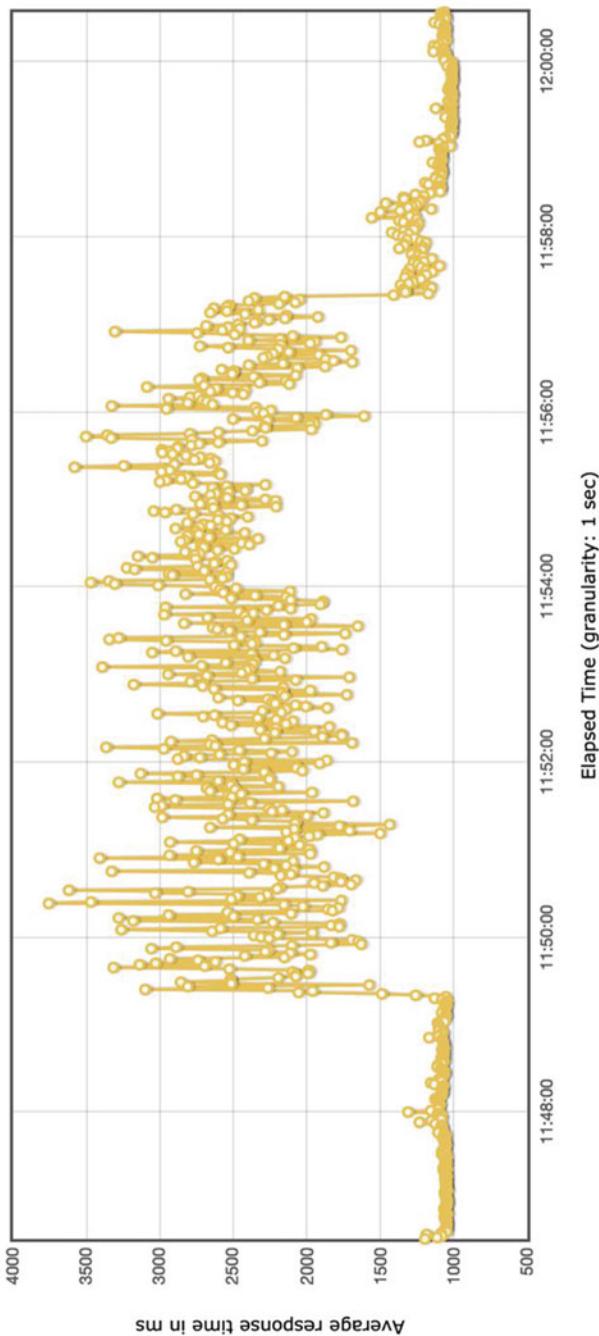


Fig. 8.16 Response time when TPS=60/s in burn CPU experiment

shows the response time when the TPS increases to 60 and the CPU occupancy is 100%. We find that there are some requests that can no longer be returned within 3s when the TPS reaches 60.

Therefore, we can conclude that when the CPU load is very high, if we want to ensure the customer's business is not affected, we need to limit the requests number of the system. As for how high to limit the requests, it needs to be obtained through several experiments. We can also observe from Figs. 8.15 and 8.16 that the response time also recovers quickly after the CPU load is restored.

7. Expanding the scope

Once we have gained some experience and reference data from small-scale experiments, we can expand the scope of the experiments. Expanding the scope of the experiment helps to discover problems that are difficult to find in small-scale experiments. For example, take the experiment mentioned in Step 1; we can expand it to experiment for two interfaces or even for multiple interfaces to observe the impact. You can also choose to perform resource exhaustion experiments on the upstream or downstream services.

The system will not become more resilient with one chaos experiment, but each experiment is a good opportunity to discover the weaknesses of the system. The team members will also grow in confidence in the system from one chaos experiment to another. After running chaos experiments iteratively, we can eventually build resilient systems.

8.3.3 *Fault Injection Experiments with System Resources*

The cloud-native platform provides theoretically unlimited scalability for applications, but, in the real world, there are resource limitations on everything, from large instances to small containers. Common system resources include CPU, memory, network, disk, etc. This section will introduce how to do fault injection of system resources in containers in a cloud-native environment based on our practice.

1. System resource-based fault injection tools

In actual development, almost all the system resource failures we encounter are the resource exhaustion types, such as CPU overload and memory burst. There are quite a few mature tools that can be used to simulate these failures, such as the commonly used tools stress and stress-ng. stress-ng is an enhanced version of stress and is fully compatible with stress. stress-ng supports hundreds of parameters, and configuring these parameters can generate all kinds of stress. We suggest using stress-ng. Stress-ng has too many parameters to list completely; here are some parameters that are often used in chaos experiments (the first half of the parameters are in abbreviated form, and the second half are written in full).

- (a) -c N, --cpu N: N indicates the number of processes started; each process will occupy one CPU when the number exceeds; the processes will compete.

- (b) -t N, --timeout N: N indicates the timeout time, beyond which the process will exit automatically.
- (c) -l N, --cpu-load N: N indicates the pressure to be applied to the CPU; -l 40 indicates 40%.
- (d) -m N, --vm N: N indicates the number of processes started, and the system will continuously allocate memory for each process.
- (e) --vm-bytes N: N indicates the size of memory allocated to each process. For example, --vm 8 --vm-bytes 80% means that 8 memory consuming processes are started, consuming a total of 80% of the available memory, and each process will consume 10% of the memory.
- (f) -i N, --io N: N indicates the number of processes started, each of which will continuously call the sync command to write to disk.
- (g) -B N, --bigheap N: N indicates the number of processes started, each process consumes memory quickly, and the system will kill them when memory is exhausted.

For example, a parameter to simulate CPU load to 40% would be stress-ng -c 0 -l 40, and a parameter to simulate memory consumption of 80% and lasting 1 h would take the form of stress-ng --vm 8 --vm-bytes 80% -t 1h.

Using stress-ng is possible to perform fault simulation for CPU, CPU-cache, devices, I/O, files, memory, and many other resources. In addition to using such tools, sometimes it is possible to develop some tools on your own in order to integrate more easily with other systems within your organization. For example, the following code shows a function that we developed ourselves—consuming CPU by constantly performing calculations:

```
package main

import (
    "runtime"
    "github.com/spf13/cobra"
)
var (
    bigNumber = 2147483647
)
func getRunCmd() *cobra.Command {
    var number int
    var runCmd = &cobra.Command{
        Use:   "run",
        Short: "start to burn CPU",
        Run: func(cmd *cobra.Command, args []string) {
            runtime.GOMAXPROCS(number)
            for i := 0; i < number; i++ {
                go func() {
                    for {
                        for i := 0; i < bigNumber; i++ {
                            }
                        runtime.Gosched()
                    }
                }()
            }
            select {} // wait forever
        },
    }
    runCmd.Flags().IntVarP(&number, "number", "n", runtime.NumCPU(), "how many logical CPUs you want to burn")
    return runCmd
}
```

2. How to conduct fault injection experiments in the container

Because of the immutable infrastructure characteristics of the container, fault injection in the container cannot modify the business code. I have used three ways in practice: one is to copy the fault injection tool into the container and execute it directly inside the container; the second is to access the container's namespace from the remote side through the nsenter command and then execute the fault injection tool remotely; the third is to package the fault injection tool into an image and deploy the tool and the business into the same container in Sidecar mode.

Below, we describe the first two approaches in detail. The third approach is only slightly different from the first in terms of deployment which can be found in Chap. 9 and is not described in detail here.

- (a) Copy the fault injection tool into the container and execute it directly in the container.

Copy the executable tools directly into the container via the Kubernetes command in the following way:

```
# Copy Fault Injection Tool
kubectl cp ./burncpu fw25048a/target-spa-http-7d8cc7b7cd-psdmz:/opt/ -c target
# burncpu is a fault injection tool that simulates CPU exhaustion
# fw25048a is a namespace in a Kubernetes cluster
# target-spa-http-7d8cc7b7cd-psdmz is the name of the Pod
# target is the name of the container
# Enter the container
kubectl exec -it -n fw25048a target-spa-http-7d8cc7b7cd-psdmz --container target /sh
# Execute the fault injection tool
cd /opt && ./burncpu
```

With these few commands, it is possible to perform fault injection experiments with CPU running at full load in the business container.

You can also call the Kubernetes API to operate the fault injection tool, such as using the remotecommand tool in the Kubernetes client-go development library to execute commands remotely, as shown in Fig. 8.17. The copy, execute, and stop operation methods are similar, with the difference being that the incoming commands are different.

The following Golang sample code shows how to implement fault injection using the remotecommand tool:

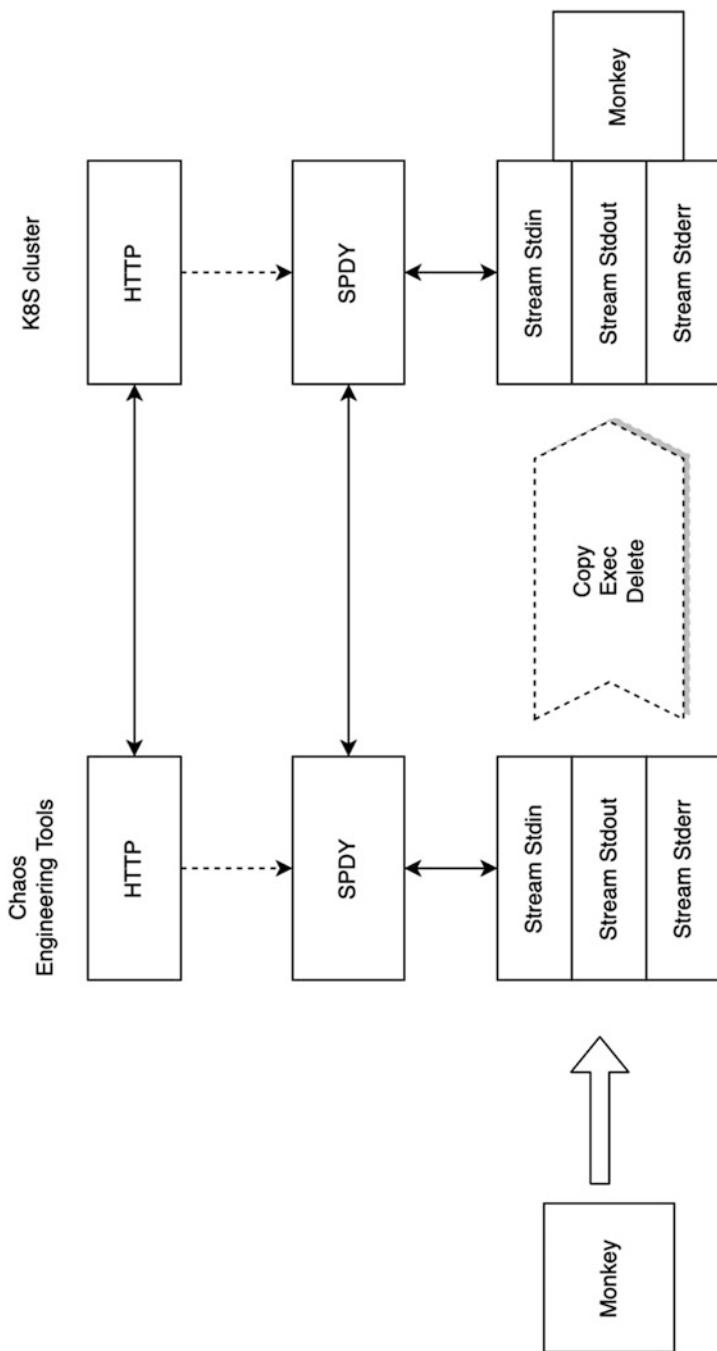


Fig. 8.17 Fault injection tool example

```

import k8s.io/client-go/tools/remotecommand

func (*DefaultRemoteExecutor) Execute(method string, url *url.URL, config *restclient.Config, stdin io.Reader, stdout, stderr io.Writer, tty bool,
terminalSizeQueue remotecommand.TerminalSizeQueue) error {
    exec, err := remotecommand.NewSPDYExecutor(config, method, url)
    if err != nil {
        return err
    }
    return exec.Stream(remotecommand.StreamOptions{
        Stdin:     stdin,
        Stdout:    stdout,
        Stderr:    stderr,
        Tty:       tty,
        TerminalSizeQueue: terminalSizeQueue,
    })
}

```

The method represents HTTP methods such as POST, GET. The URL needs to contain namespace, resource, container, and command inside. config is the kubeconfig object, which is the key used to access the Kubernetes cluster. The following pseudo code shows how to get the URL:

```

import (
k8sCorev1 "k8s.io/api/core/v1"
k8sSchema "k8s.io/client-go/kubernetes/scheme"
)
cmds := []string{/opt/bin/burncpu"}
baseReq := coreclient.RESTClient().
Post().
Namespace(i.Namespace). // namespace
Resource("pods"). // Resource, e.g. pods
Name(i.Name). // resource name, e.g. Pod name SubResource("exec")
execReq := baseReq.VersionedParams(&k8sCorev1.PodExecOptions{
Container: containerName,
Command: cmds,
Stdin: true, Stdout: true,
Stderr: true,
TTY: false, }, k8sSchema.ParameterCodec)
// Get the URL
url := execReq.URL()

```

This approach is well suited for scenarios where fault injection tools are integrated into chaos engineering platforms.

- (b) Access the container's namespace from the remote side via the nsenter command, and then execute the fault injection tool remotely.

The following is an example of how to use the nsenter command for our attempts in chaos experiments. As shown in Fig. 8.18, we deploy a Chaos Experiment DaemonSet service in the cluster, through which we perform fault injection experiments on other business Pods in the cluster. The image for this service is based on a Linux image; for example, you can choose the debian:buster-slim Docker image. nsenter is in the util-linux package and can

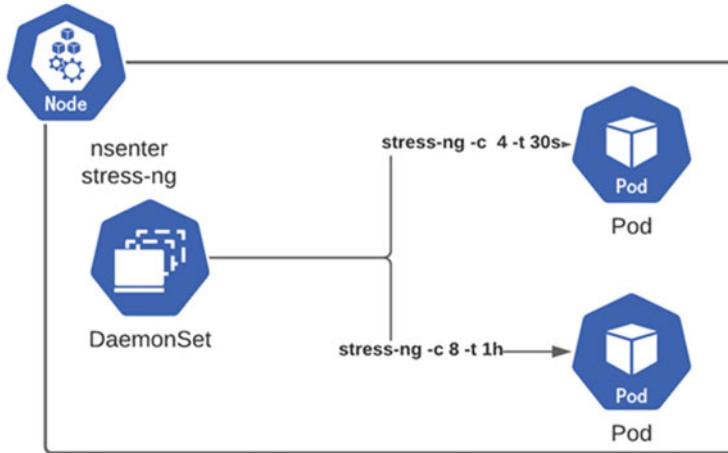


Fig. 8.18 nsenter in chaos experiment

be used directly. When packaging, you can install stress-ng to facilitate subsequent fault injection experiments.

```
# Dockerfile example
FROM debian:buster-slim
RUN apt-get update && apt-get install -y stress-ng
```

You can refer to the deployment method in Chap. 9 to deploy the DaemonSet service. Next, enter the DaemonSet container via the Kubernetes command and execute the nsenter command in the container. As shown below, the CPU load is 60% for 20s:

```
nsenter -t 99258 stress-ng -c 0 -l 60 --timeout 20s --metrics-brief
# -t 99258 is the PID of the container
```

The method to get the container PID in a Kubernetes cluster is as follows:

```
# Get the container ID, i.e. containerID
kubectl get pod podName -o template --template='{{range .status.containerStatuses}}{{.containerID}}{{end}}'
# Get PID by containerID
docker inspect -f {{.State.Pid}} containerID
```

This approach is particularly suitable for scenarios where fault injection tools are integrated into chaos engineering platforms.

3. How to integrate fault injection tools into chaos engineering platforms

We usually integrate fault injection tools into the Chaos Engineering Platform and start and stop experiments through the platform. Two methods for building a chaos engineering platform are described below for the reader's reference.

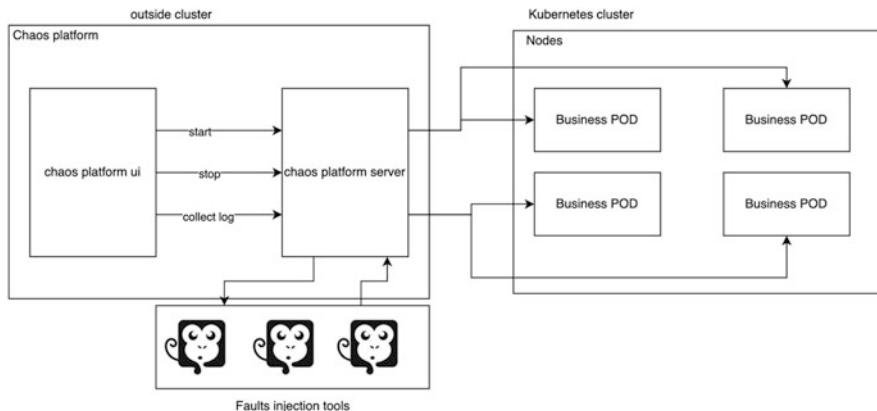


Fig. 8.19 Plugin mode chaos engineering platform

(a) Plug-in mode of chaos engineering platform

The fault injection tool and the chaos engineering platform are independent, and this architecture is suitable for the chaos engineering platform that needs to be isolated from the business cluster. The Chaos Engineering Platform UI is mainly responsible for interactive operations, including starting and stopping experiments. The Chaos Engineering Platform Server is mainly responsible for receiving requests from the UI, selecting the appropriate fault injection tool according to the requests, copying the tool into the business container, and executing experiments.

We use the remotecommand tool described above to implement the interaction between the Chaos Engineering Platform Server and the Kubernetes cluster. The fault injection toolset is a collection of the various tools mentioned above, which is a set of executable binaries. The Chaos Engineering Platform Server gets the operation of each tool through a configuration file. The whole structure is shown in Fig. 8.19.

(b) Building chaos engineering platform through cloud-native approach

The principle of building a chaos engineering platform through a cloud-native approach is shown in Fig. 8.20. This way comes from PingCAP team and is built based on the custom resource CRD of Kubernetes. Various chaos experiments can be defined by the custom resource CRD, such as chaos experiments for Pods, network chaos experiments, etc.

In the Kubernetes ecosystem, CRD is a mature solution for implementing custom resources and has very mature supporting tools such as the kubebuilder tool. This tool can be used to automatically generate the code to manage CRD instance controllers. The chaos experiment controller in Fig. 8.20 was generated by kubebuilder. This chaos engineering platform contains the definitions of various chaos experiment objects generated by CRD, the chaos engineering platform UI, the chaos experiment controller, and the chaos experiment DaemonSet service.

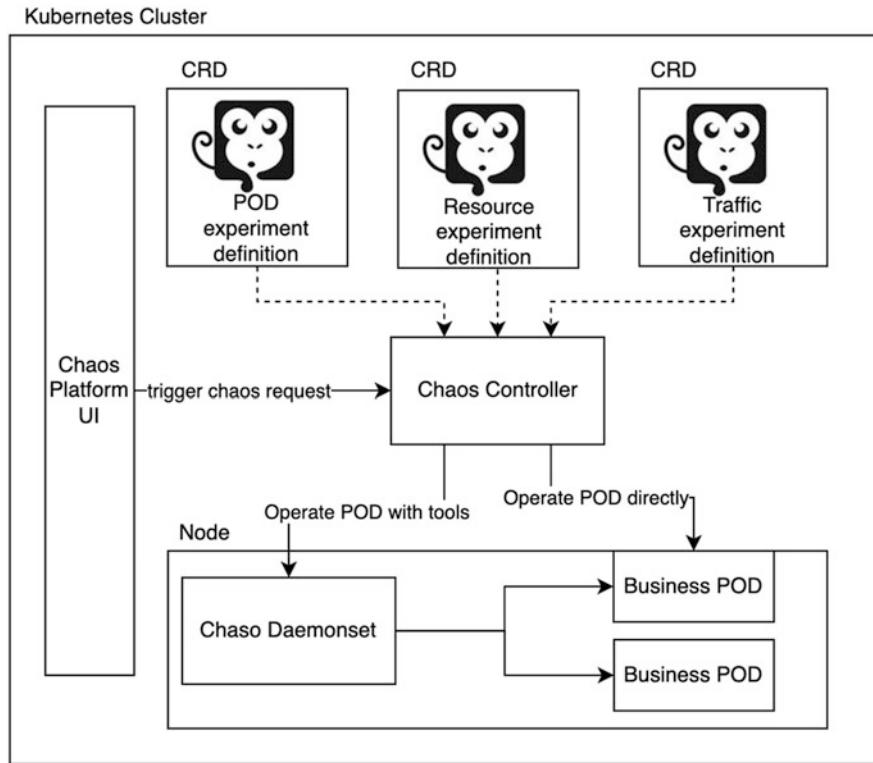


Fig. 8.20 Cloud-native chaos engineering platform

The Chaos experiment object definition file is a yaml file, which defines the parameters needed for each type of experiment, as shown in the following resource fault injection experiment definition. The Chaos Engineering Platform UI is used to interact with the user to add, delete, and check the experimental objects. The Chaos Experiment Controller is used to accept requests from the Chaos Engineering Platform UI, transform the requests into Chaos Experiment instances, and then run the Chaos Experiment. If the experiment is performed directly on a Pod, it interacts directly with the business Pod, such as a kill Pod. If you want to run a chaos experiment on a container inside a Pod, you need to send a request to the Chaos Experiment DaemonSet service on the corresponding Node node, and the DaemonSet service will complete the chaos experiment. For example, to run an experiment of memory occupation inside a container, remote operation of the container memory can be done by the DaemonSet service through nsenter.

The following code shows how to define the fault injection experiment object:

```
# Fault injection experiment object definition, intercepted some important fields
---
apiVersion: apiregistration.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.2.5
  name: stresschaos.fw.bsap.chaos
spec:
  group: fw.bsap.chaos
  names:
    kind: StressChaos
    scope: Namespaced
  validation:
    openAPIV3Schema:
      description: StressChaos is the Schema for the stresschaos API
  properties:
    apiVersion:
      type: string
    kind:
      type: string
    metadata:
      type: object
    spec:
      properties:
        containerName:
          type: string
        duration:
          type: string
        selector:
          properties:
            labelSelectors:
              additionalProperties:
                type: string
              type: object
            pods:
              additionalProperties:
                items:
                  type: string
                  type: array
                  type: object
                type: object
        stressngStressors:
          type: string
        stressors:
          properties:
            cpu:
              properties:
```

```

load:
  type: integer
type: object
value:
  type: string
required:
- mode
- selector
type: object

```

The following is a brief description of the implementation of the fault injection experiment, using Chaos Mesh as an example.

First, prepare the experimental environment. Chaos Mesh requires Kubernetes version 1.12 or above and open RBAC functionality. You need to build an eligible Kubernetes cluster and deploy business services to the cluster.

Then, install Chaos Mesh. It is recommended to use Helm3 to deploy Chaos Mesh with the following code:

```

# Add Chaos Mesh repo
helm repo add chaos-mesh https://charts.chaos-mesh.org
# Create namespace, deploy Chaos Mesh to chaos-testing namespace
kubectl create ns chaos-testing
# Install Chaos Mesh using Helm3
helm install chaos-mesh chaos-mesh/chaos-mesh --namespace=chaos-testing
# Check if Chaos Mesh is installed successfully
kubectl get pods --namespace chaos-testing -l app.kubernetes.io/instance=chaos-mesh

```

Finally, run the Chaos experiment. Once Chaos Mesh is installed, you can directly define the experiment object and create the experiment by using the kubectl command with the following code:

```

# forecast-network-delay.yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
  name: forecast-delay
spec:
  action: delay # Perform network delay experiment
  mode: one # mode: one # Pod at a time
  selector: # Pod selection conditions
  namespaces:
    - fw
  labelSelectors:
    "app": "forecast" # Label on business Pod
  delay:
    latency: "10ms" # Network delay time 10ms
    duration: "30s" # network delay lasts 30s
    scheduler: # execute every 60s cron
    cron: "@every 60s"
  kubectl apply -f forecast-network-delay.yaml -n chaos-testing

```

If you encounter problems during the experiment, you can simply stop the experiment with the following command:

```
kubectl delete -f forecast-network-delay.yaml -n chaos-testing
```

Chaos Mesh also provides the ability to manipulate experiments directly from the UI, as shown in Fig. 8.21. You can click the “+ NEW EXPERIMENT” button to create chaos experiments, such as creating experiments that simulate Pod failures or network failures. You can also click the “+ NEW WORKFLOW” button to schedule chaos experiments for the purpose of executing multiple experiments serially or in parallel. New users can click on the “TUTORIAL” button to learn how to use Chaos Mesh.

System resource failure is a relatively common failure. When we encounter this kind of problem, our first reaction is to restart the business process. By simulating such failures in advance through chaos experiments, we can find a solution plan and reduce the number of blind restarts of services, which can make the services more stable and reliable.

8.3.4 Service Mesh-Based Network Traffic Fault Injection Method

In a cloud-native environment, container orchestrators can quickly fix single points of failure, such as Kubernetes, by creating new Pods to replace the failed Pods and keep the business up and running. However, along with the development of microservices, the dependencies between services become intricate and complex, and the probability of failure in connectivity between services increases. Kubernetes cannot automatically fix such problems, so we need to simulate such failures in advance to test the system’s ability to recover from errors. This section introduces network traffic failure injection method based on service mesh, using Istio and Linkerd as examples.

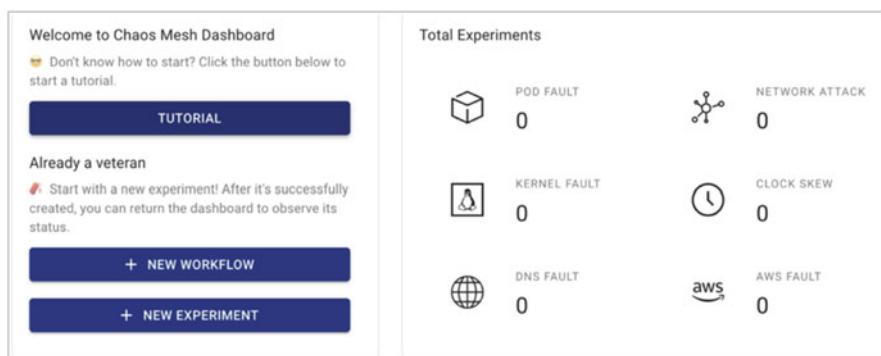


Fig. 8.21 Chaos-mesh

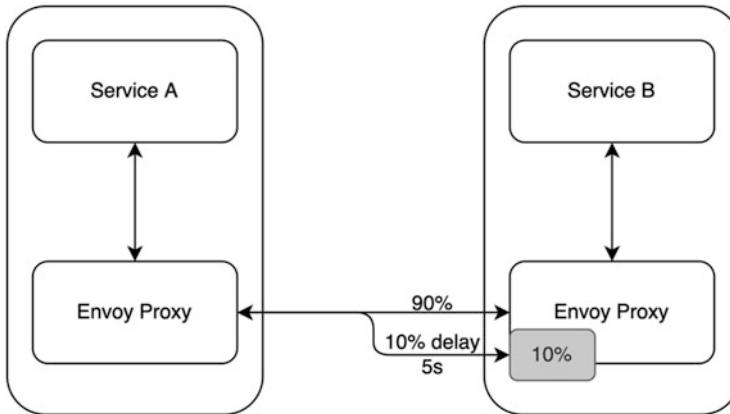


Fig. 8.22 Istio-based fault injection

1. Istio-based network traffic fault injection method

The principle of using Istio can be found in Chap. 4. This section only introduces Istio's network traffic fault injection method, as shown in Fig. 8.22. Service A sends requests to Service B. Network traffic fault injection is achieved by taking traffic through Service B's Envoy Proxy.

Istio provides two types of fault injection, network latency, and network outage. Network latency mainly simulates network delays or overloads, and network outage mainly simulates upstream service crashes. Both types of fault injection are implemented by configuring the HTTP node in Istio's Virtual Service.

(a) Configuration of simulated network delay

Let's start with the configuration of the simulated network delay, with the parameters described in Table 8.7.

An example of network latency is as follows, with a 5s delay occurring in one out of every 100 requests:

```

fault:
delay:
percentage:
  value: 1
fixedDelay: 5s

```

(b) Configuration of simulated network interruptions

Then, let's introduce the configuration of network interrupt, with the parameters described in Table 8.8.

The network outage example is as follows, with one out of every 100 requests returning a 404 status code:

Table 8.7 Delay parameters

Parameter	Description
fixedDelay	Fixed delay time before Envoy forwards traffic to the application, in the format 1h/1m/1s/1ms, with a minimum value of 1 ms, is a required field
percentage	The percentage of traffic shifting that experiences delay is in the range [0.0, 100.0]. 0.1 which means that 0.1% of requests incur network delays

Table 8.8 Abort fault parameters

Parameter	Description
httpStatus	HTTP status code of the interrupted request
percentage	The proportion of requests where interruptions occur, range [0.0, 100.0]

```

fault:
abort:
percentage:
value: 1
HttpStatus: 404

```

Both types of fault injection can be achieved by configuring http.match in Virtual Service. Table 8.9 lists the common configuration parameters used in fault injection.

The following is an example of using Istio to configure delayed fault injection in practice, which delays requests sent to adunit.fw43320.svc.cluster.local with the prefix "/list" by 7s:

```

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: network-delay
labels:
use.istio: "true"
spec:
hosts:
- adunit.fw43320.svc.cluster.local
http:
- match:
- uri:
prefix: /list
fault:
delay:
percentage:
value: 100.0
fixedDelay: 7s
route:
- destination:
host: adunit.fw43320.svc.cluster.local
port:
number: 3450

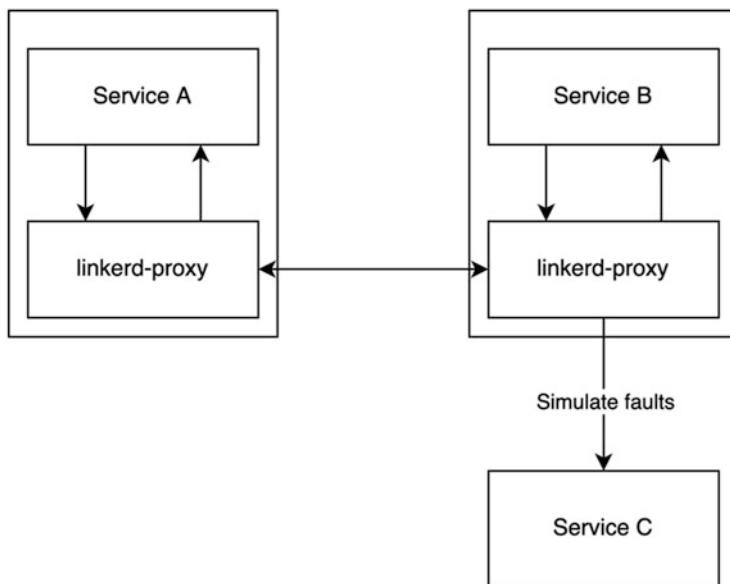
```

Table 8.9 Request matcher parameters

Parameter	Description
uri	There are three ways to match URIs: exact match, prefix match, and regex regular match. For example, prefix: /foo matches URI requests prefixed with foo
authority	Match HTTP Authority value in the same way as URI match URI
headers	Match HTTP Header values in the same way as the URI, e.g., :headers: x-fw-network-id: exact: 57230
queryParams	Matching request parameters, such as ?id=12, using the same method as matching headers, but matching only supports two kinds of matching methods: exact and regex
withoutHeaders	Contrary to the meaning of headers, match requests that do not contain this configuration

2. Linkerd-based network traffic fault injection method

Linkerd is also a service mesh solution that differs from Istio in its fault injection approach. Linkerd uses the Traffic Split API to simulate network faults. Figure 8.23 shows how failure simulation is performed through Linkerd, where traffic is split from Service A to Service B. In Service B, a portion of the traffic is split and forwarded to the failure simulation service through Linkerd Proxy.

**Fig. 8.23** Linkerd-base fault injection

Using this approach for network traffic fault injection is a two-step process: first, creating a fault simulation service in the cluster, and, second, creating a traffic split resource (TrafficSplit).

(a) Create a fault simulation service in the cluster

The failure simulation service can be chosen on demand. The official documentation for Linkerd uses Nginx as the failure simulation service. Regardless of the service used, there are two caveats: first, Linkerd's splitting of traffic occurs at the Kubernetes service layer, and the Service object of the service needs to be deployed correctly; second, you need to specify the subsequent behavior of the request received by the service. Let's look at an example.

The Nginx configuration, which returns a 500 status code after receiving a request, has the following code:

```
http {  
    server {  
        listen 8080;  
        location / {  
            return 500;  
        }  
    }  
}
```

The service object configuration for Nginx in a Kubernetes cluster is as follows:

```
apiVersion: v1  
kind: Service  
metadata:  
  name: network-chaos  
  namespace: chaos  
spec:  
  ports:  
  - name: service  
    port: 8080  
  selector:  
    app: network-chaos
```

(b) Create traffic segmentation resources

By creating a traffic split resource (TrafficSplit) in the cluster, Linkerd can then forward a portion of the traffic to the failure simulation service. The configuration code is as follows. Linkerd receives requests from adunit, sends 90% of the requests to the adunit service, and sends 10% of the traffic to the Nginx fault simulation service created in the first step:

```

apiVersion: split.smi-spec.io/v1alpha1
kind: TrafficSplit
metadata:
  name: error-split
  namespace: chaos
spec:
  service: adunit
  backends:
    - service: adunit
      weight: 90
    - service: network-chaos
      weight: 10

```

Both network traffic fault injection methods described in this section are based on application layer implementations. In Istio, Envoy can directly respond to requests with delay and stop, while in Linkerd a third service is required to achieve this:

8.4 Quality Assurance of Production-Like Environments

Traditionally, software testing mainly refers to local test results, but, as applications go to the cloud, local test results are increasingly difficult to represent the actual state of the production environment. Therefore, in recent years, the topic of quality assurance for production environments has gained more attention, such as blue-green deployment, canary release, and disaster recovery, which are some of the attempts made by the industry to ensure the quality of production environments.

Some of FreeWheel's customers choose to test in a pre-release environment (production-like environment), so it is same important for our team to ensure the stable availability of the pre-release environment as it is to ensure the stability of the production environment; both environments have the following characteristics compared to the local one:

- Realistic user behavior: Both types of environments are used by real users and thus many test behaviors are limited to take, e.g., stress testing may result in the system being unusable.
- The systems are more complete: When testing locally, many external dependencies may be resolved using mock, but the products in these two types of environments are complete products integrated with multiple systems, and the test environment is much more complex than the local test environment.
- High data complexity: Online data originates from real user businesses and is far more complex than the data simulated by the test environment.
- Access restrictions: Online systems have specific security and audit requirements, and some servers and data are not directly accessible, making it difficult to troubleshoot problems.

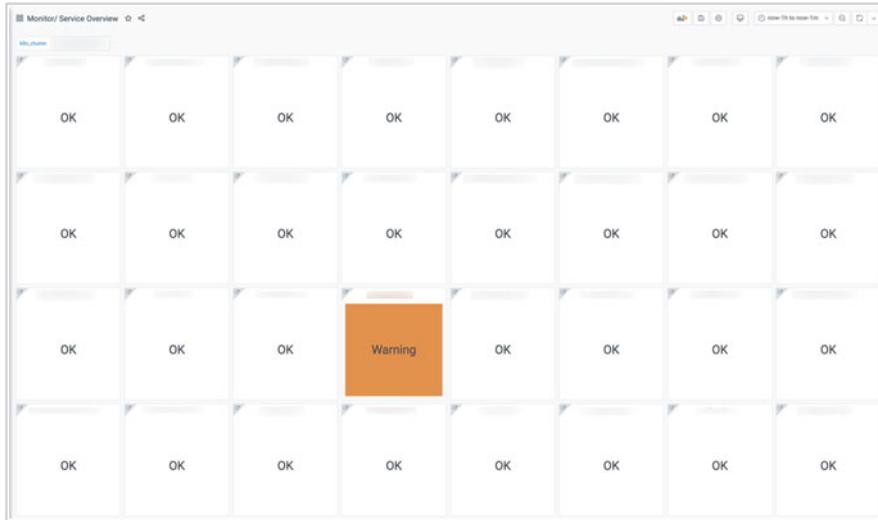


Fig. 8.24 A grafana dashboard page for micro-services overview

- Long feedback cycle: Changes by the development team need to go through a full round of testing cycles before they finally reach the customer.

These characteristics dictate a different set of solutions to be used when safeguarding product quality in production-like environments. In this section, some specific practices will be presented.

8.4.1 *Monitoring and Analysis of Online Services*

For the characteristics of the online environment, we need to adopt different strategies to achieve the quality assurance of the system as follows.

1. Monitoring and alarming

In Chap. 7, we introduced the observability of services. By collecting logs of services, tracing requests, and defining metrics, we can get a clear picture of the real-time state of online services. For production-like environments, we can customize the corresponding monitoring dashboard according to our requirements to have an intuitive understanding of the state of the online environment.

As shown in Fig. 8.24, the dashboard shows the real-time status of the online microservices, allowing the engineer to have an initial understanding of the real-time status of the entire system.

2. Page Analysis Tools

For client-side products, some specific page analysis tools can also be applied to collect usage data of online services. For example, in our team, we have

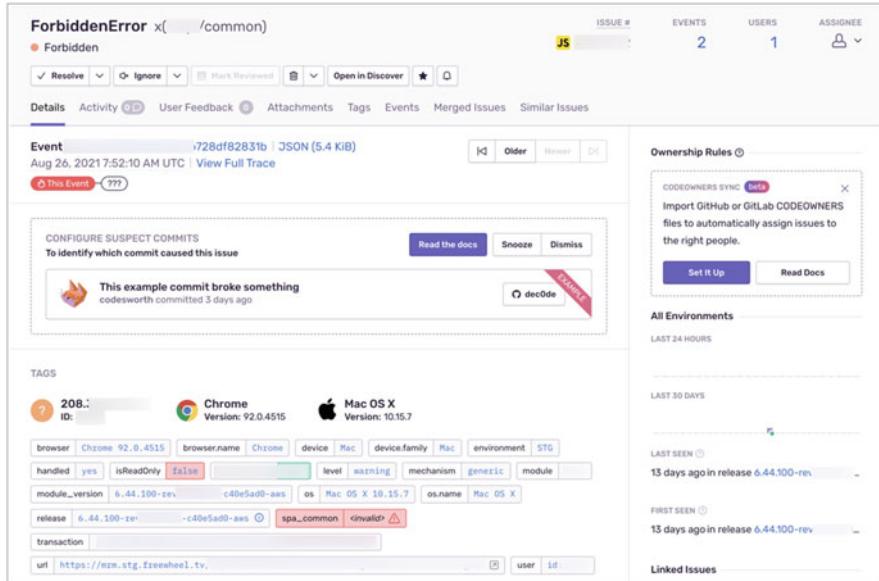


Fig. 8.25 A frontend error Sentry UI page example

introduced the Sentry integration for real-time monitoring and feedback on the usage of the website pages. Sentry is mainly used for error reporting, monitoring, analysis, and alerting, and its feedback interface is shown in Fig. 8.25. The error message contains information such as error description, call stack, program running environment, network request, and user identification.

3. User feedback

Clients, as users of the system, are more sensitive to the perception of changes to the system. Feedback based on user behavior has the following effects.

- (a) It can help the team to design more sensible test cases.
- (b) The scope of impact of the current changes can be more precisely defined.
- (c) It will allow the system to better meet user needs in subsequent improvements.

4. Testing diversity

From a layered perspective, online environment testing is the closest to end-to-end testing, so we do not recommend writing too many automated test scripts to implement online environment testing. In contrast, the production-like environments can use methods like Bug Bash and Post-release Check for quality assurance based on their special characteristics.

8.4.2 Bug Bash Practice

Bug Bash is a test engineering practice that the team has been running since 2015 and is held before each new release goes live. Each microservice business development team holds a Bug Bash before go-live, inviting developers from other groups to participate in order to find issues that are hard to detect by functional developers. At the end of the event, participants are judged on the number and quality of bugs found and given some material rewards (e.g., coffee vouchers).

Bug Bash has become an important part of the team's development process, and our product quality has improved as a result, with several serious bugs being found and fixed during this activity before releasing to clients. Figure 8.26 illustrates the main stages of Bug Bash. The below section will describe the practice of Bug Bash according to the phases in Fig. 8.26.

1. Preparation of activities

Products are deployed to a pre-release environment and go through 2 weeks of testing before being deployed to production. During this time, the various business teams conduct Bug Bash activities in about 2-h meetings.

For large projects that span multiple departments, consider inviting colleagues from other departments to come along and find bugs.

- (a) Helped find a lot of unexpected bugs.
- (b) Increase everyone's business knowledge and understanding of business functions outside their own scope of work.
- (c) Promote team building and improve communication between colleagues from different business teams and departments.

In addition to the above arrangements, each business team will need to complete the following preparations prior to the event taking place.

- (a) Test environment and data preparation: There are some features that need to be turned on in advance, and data also needs to be prepared so that the test can be conducted.
- (b) Test points listing: Each version will have different changes, and each business team should prepare its own list of changes to help colleagues in the event clarify the main content and direction of the test.
- (c) Bug Identification Rule Definition: Identify a clear set of criteria to help participating colleagues clarify what types of issues are considered bugs.

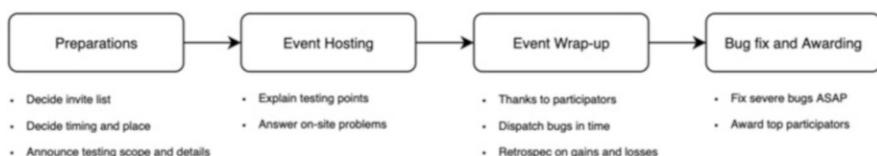


Fig. 8.26 Bug Bash process

Ticket	Priority	Summary	Reporter	Assignee	Status	Fix Versions	Components
FW-70	Should Have	The message for update operations is disappeared automatically	Guo	Ren	Done	6.44	Biz - UI - Programmatic
FW-7C	Should Have	[Schedule Builder] ad industry export view should not display "[]".	Xu	Chen	To Do	6.45	Biz - UI - HyDA & Linear Schedule Ingest
FW-7C	Should Have	[SAI] export file name should cover channel along with UI change	Xu	Jiang	Done	6.44	Biz - UI - Inventory

Fig. 8.27 A bug bash report page example

2. Event hosting

Each business team's Bug Bash event is held as a meeting, and a person will be assigned to host it. The moderator's main job consists of two things.

- (a) Explain this go-live project and important test points.
- (b) Answering questions from other participants in the meeting.

In addition, other members of the business team need to keep an eye on the activity and help other participants in a timely manner to confirm the bug situation, reproduce the bug, etc.

3. Event wrap-up

After each event, each business team needs to schedule an internal meeting to wrap up the Bug Bash event by completing the following pieces of work:

- (a) The severity classification of each bug and duplicate bugs need to be closed.
- (b) A schedule is made for fixing each bug, and very serious bugs need to be fixed before going live; Fig. 8.27 shows an example list of bugs that need to be fixed for an event.
- (c) Retrospect on the gains and losses of the event.

4. Bug fix and awarding

Bug Bash Tools was created to provide statistics on the results of Bug Bash activities. It counts the bugs found according to different metrics and aims to help improve product quality by analyzing this data. The following are the specific metrics.

(a) Single-result statistics

Figure 8.28 shows the statistics for a single Bug Bash campaign. We will analyze the effect of this campaign in three dimensions:

- (i) The pie chart in the top left corner shows a comparison of the severity of the bugs identified. The main thing to look at here is the weight of Must Have and Should Have; if the proportion of bugs in these two categories is too high, it means that the early stages of quality assurance are not done well enough and needs to be reflected on and adjusted.

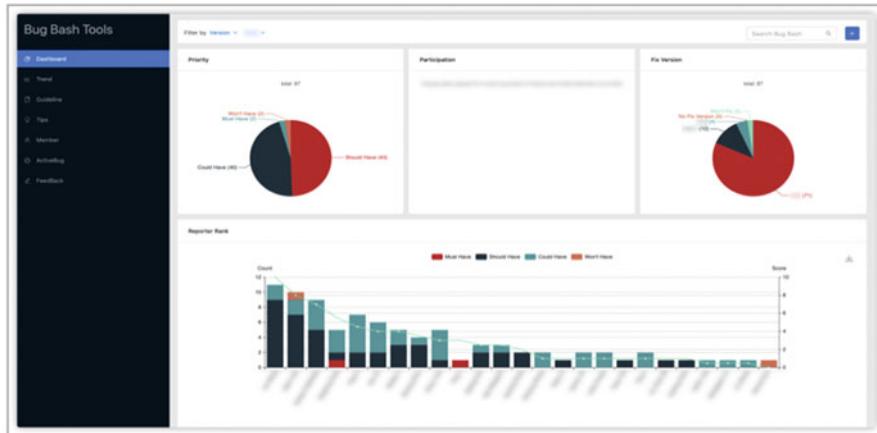


Fig. 8.28 A bug bash dashboard page overview example

- (ii) The pie chart in the upper right corner assesses the urgency of the fixes for the identified bugs in terms of the percentage of time it took to fix them. If the percentage of fixes for the current version submitted after the campaign is high, it means that the campaign found a lot of pressing issues. On the plus side, this means that we are finding issues that need to be fixed before customers use new versions of the product; but on the other side, it also reflects that the early stages of quality assurance are not done very well.
 - (iii) The bar chart at the bottom reflects a ranking. Each participant's bugs are weighted according to their severity, and a ranking bar chart is created based on their score. The team will reward the top-ranking colleagues to recognize their motivation and encourage more people to find bugs.
- (b) Trend statistics

In addition to the single result statistics, we will also combine the results of multiple rounds to try to find some noteworthy conclusions from the trending. Figure 8.29 shows the trend in the severity of bugs identified in dozens of campaigns over the past 4 years. It also shows that our team went through a painful period during the back-end service transformation and front-end single-page application transformation, but the overall number of bugs has been decreasing over time.

After so many Bug Bash campaigns, we've learned a few lessons.

- (i) Bug Bash is not a mandatory event for each business team. Each team should arrange the frequency of the event according to its own situation.

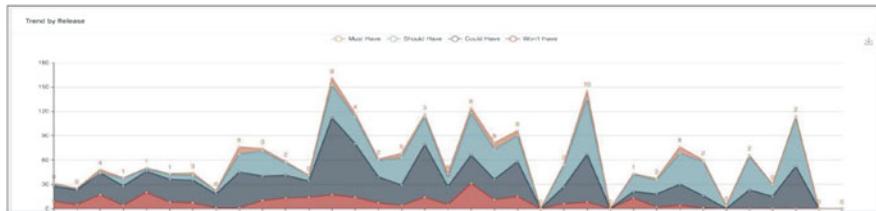


Fig. 8.29 A bug bash trending page example

- (ii) There is no need to stick to offline activity; as long as the campaign materials are well prepared, you can also choose to hold an online Bug Bash so that more of your colleagues can join in.
 - (iii) Pay attention to bug recording. For the problems found in the meeting, use video recording, screenshots, and other ways to record in a timely manner and to facilitate later developers to analyze and investigate; otherwise when bug reproduction steps are not clear, the corresponding team cannot define the severity of the bug and other problems easily.

8.4.3 Post-release Check Practice

After the product is successfully deployed to the online environment, it does not mean that everything is fine. It is still necessary to check online functionalities at this time, also known as Post-release Check (use Post Check in short).

1. Basic principles

In the previous section, we have already done very detailed testing, so this stage is aimed to cover as many product features as possible at the lowest cost; in other words, cover the optimistic route (Happy Path) as the top priority to ensure that the user's basic functions are not hindered and other routes, as a lower priority, including pessimistic route testing, stress testing, etc.

2. Automation

As the process of moving to micro-service continues, automated test cases for post-release checks are gradually being unified within the Cypress framework system for end-to-end testing. There are two steps to finish the post-release check practice.

- (a) Tied to a continuous deployment pipeline: triggered automatically with every online deployment, saving effort for triggering and regression testing.
 - (b) Differentiate the priority of test cases: Test cases are divided into two different levels, P1 and P2. For P1 test cases, once they do not pass the test,

the person in charge of the corresponding test case is required to immediately check the reasons and reply to them.

Figure 8.30 shows the result email that triggers the pipeline after one deployment.

3. Data statistics

In addition to presenting the results of a single deployment, we can save test results into the database and present results through a Post Check Tool, as shown in Fig. 8.31. Doing so brings two benefits:

- Analyze historical data trends to identify potential problems: Traditional business test cases are relatively stable, so we can catch possible problems by observing changes in the trend of the same test case. For example, if a test case has taken 30% longer to run successfully in the last few releases than the previous average, it is likely that the changes in the last few releases are the cause.
- Identify unstable test cases: as an end-to-end test, there is a certain probability that a test link will fail because it is unstable. However, if a test case often fails to run, there must be room for improvement—the framework itself is not stable enough, or the test case itself is not stable enough. By counting the probability of failure, you can filter out such unstable test cases and then have the corresponding test case owner improve them until they are stable.

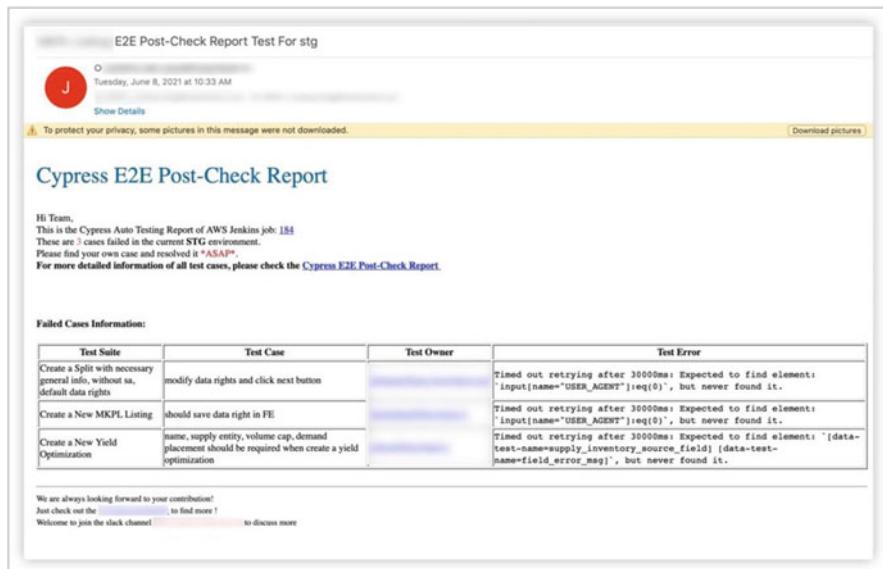


Fig. 8.30 A E2E post-check report email example

Case Summary											
Case Name	Path	Owner	%	Failure Count	Total	%	Failure Rate	Tags	%	Latest Failure	Latest Pass
MP1 Order Extension E2E Case mp1 order extension decline order extension	spec/cases/mp1/order		1	13	7.69%				2021-08-171023:39:11Z	2021-08-31107:06:49Z	
Exchange Listing Creation & Copy & Edit Create a New Exchange Listing should successfully create the exchange listing	spec/cases/mp1/listing		1	13	7.69%				2021-08-171023:34:35Z	2021-08-31107:06:37Z	
Yield Optimization Create & Edit & Delete/Update Create a New Yield Optimization create yield optimization with inventory order, assay and global control	spec/cases/mp1/settings		1	13	7.69%				2021-08-171023:34:35Z	2021-08-31107:06:37Z	
Inventory Split Create Create a Split with necessary general info, without sel, multiple data rights rules create several increased attributes in inventory step	spec/cases/mp1/listing		2	28	7.14%				2021-08-171023:12:06Z	2021-08-31107:34:13Z	
VCBS Insertion Order Form Add New Insertion Order	spec/new_order/beckinsel_order		1	14	7.14%				2021-08-241008:58:17Z	2021-08-31108:17:21Z	
Inventory Split Edit Edit the existed inventory split, inventory split by brand/endpoint edit in split step	spec/cases/mp1/listing		1	14	7.14%				2021-08-181013:33:50Z	2021-08-31107:34:13Z	
Inventory Split Create Create a Split with necessary general info, with multiple brands and endpoint owners, custom data rights should create with split by programmer and brand	spec/cases/mp1/listing		1	18	5.56%				2021-08-201015:26:24Z	2021-08-31107:34:13Z	
Check Inventory Insights Performance Tab Check if all widgets works without error	spec/cases/mp1/inventory_insights		0	2	0.00%			NEED TO CHANGE CASE OWNER		2021-08-23111:14:15Z	
Check Inventory Insights Performance Tab Check filters	spec/cases/mp1/inventory		0	1	0.00%			NEED TO CHANGE CASE OWNER		2021-08-25110:47:34Z	
Check Inventory Insights Performance Tab Check if all widgets works	spec/cases/mp1/inventory		0	1	0.00%			NEED TO CHANGE CASE OWNER		2021-08-25110:47:34Z	

Fig. 8.31 A post-check dashboard page example

8.4.4 Disaster Recovery Strategies and Practices

Disaster recovery contains two layers: backup before a disaster and recovery after a disaster. Cloud disaster recovery is a cloud-based implementation of disaster recovery that combines cloud computing, cloud storage, and many other technologies to achieve disaster recovery, reducing enterprise investment in IT infrastructure and providing faster response time and higher recovery quality than traditional disaster recovery solutions.

1. Basic concepts

Before determining a disaster recovery strategy and solution, two key metrics need to be defined: the Recovery Time Objective (RTO) and the Recovery Point Objective (RPO).

- (a) *Recovery Time Objective*: The length of time between the occurrence of a disaster and the point in time when the system is fully recovered. The smaller this value is, the faster the system recovers.
- (b) *Recovery Point Objective*: The crashed IT system can be recovered to some historical point in time; this time point is the recoverable data history time point; there is a part of actual data loss between that time point and the disaster, so this indicator reflects the degree of data recovery integrity. The smaller this value is, the lesser business data is lost.

Figure 8.32 illustrates the relationship between these two indicators.

2. Basic strategies

Disaster recovery is generally divided into three levels: data level, application level, and business level. Among them, data-level and application-level disaster recovery can be achieved by professional disaster recovery products within the scope of IT systems. Business-level disaster recovery is based on data level and

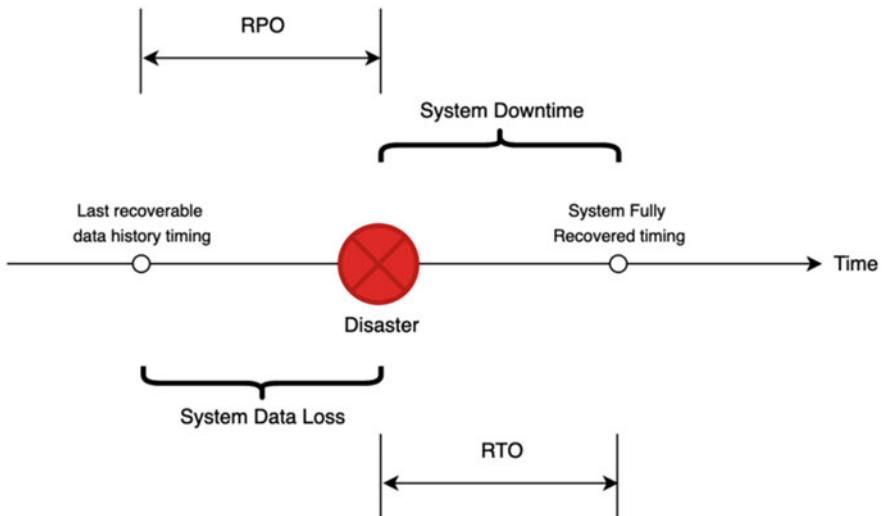


Fig. 8.32 RTO and RPO

application-level but also requires protection for factors outside the IT system, such as office location and office personnel.

As seen in Fig. 8.32, the smaller the time interval between the point of full system recovery and the occurrence of a disaster, the smaller the impact on the business, which simply means that it is important to get the system up and running as quickly as possible. The more core the business is, the more important it is to ensure that the RPO and RTO converge to zero, and the greater the corresponding investment will naturally be. Therefore, enterprises can arrange different levels of disaster recovery according to the importance of different businesses to achieve a balance between RTO, RPO, and cost.

3. Disaster recovery practices

Based on the above strategies, our team has practiced and learned some disaster recovery lessons.

(a) Performing database disaster recovery

Disaster recovery of data is divided into two parts: backup and recovery. Not all data needs to be backed up; our experience is to back up data that needs to be recovered when a disaster occurs, such as a customer application data. There are some data that can be regenerated by simple operations, and then these kinds of data do not need to be backed up. For example, the customer's daily report data can be generated by triggering the report publishing operation again.

FreeWheel's OLTP database uses AWS Aurora DB and uses a primary/replica architecture as a disaster recovery strategy. Applications in different data centers write data to the primary database, and the primary database will

The screenshot shows a Jenkins pipeline configuration page titled "Pipeline 0211_inventory-kubernetes-helm_deployment". It includes fields for "package_version" (with a redacted value), "ticket" (with values "OPS-", "OPS-", and "master"), and "pipeline_revision" (with a redacted value). There are checkboxes for "AWS_EAST" and "AWS_WEST", both of which are checked. A large blue "BUILD" button is at the bottom left.

Fig. 8.33 A Jenkins deployment page example

synchronize data to the replica database in real-time. The two databases cross data center, and, currently, we deploy them by deploying the primary database in the US East node and the replica database in the US West node. We do not use a dual-active architecture due to the large network transmission latency in the US East and US West.

In addition, FreeWheel has some data that is stored in a Key-Value type database, for example, metadata of video files, source information of websites, etc. For such data, we choose AWS's DynamoDB database, which supports the Global Table feature. Multiple data centers can be specified at the database creation stage, and the database product itself ensures that the data is synchronized. When an application writes data to a database in one region, DynamoDB automatically propagates the write operation to databases in other AWS regions.

(b) Perform stateless application disaster recovery

For stateless applications, we use an offsite multi-live strategy for multi data-center deployments. When a disaster occurs and a data center switchover happens, the application does not require additional startup time to continue to provide services to customers.

To achieve this goal, we provide easy and convenient deployment operations through a continuous deployment pipeline. Each deployment is performed for multiple data centers by default, ensuring that applications in multiple data centers run the same version of code. Figure 8.33 shows that two data centers, AWS_EAST and AWS_WEST, can be selected.

(c) Traffic switching and recovery

FreeWheel has two main official domain traffic portals: one is the company's official website and the other is the Open API.

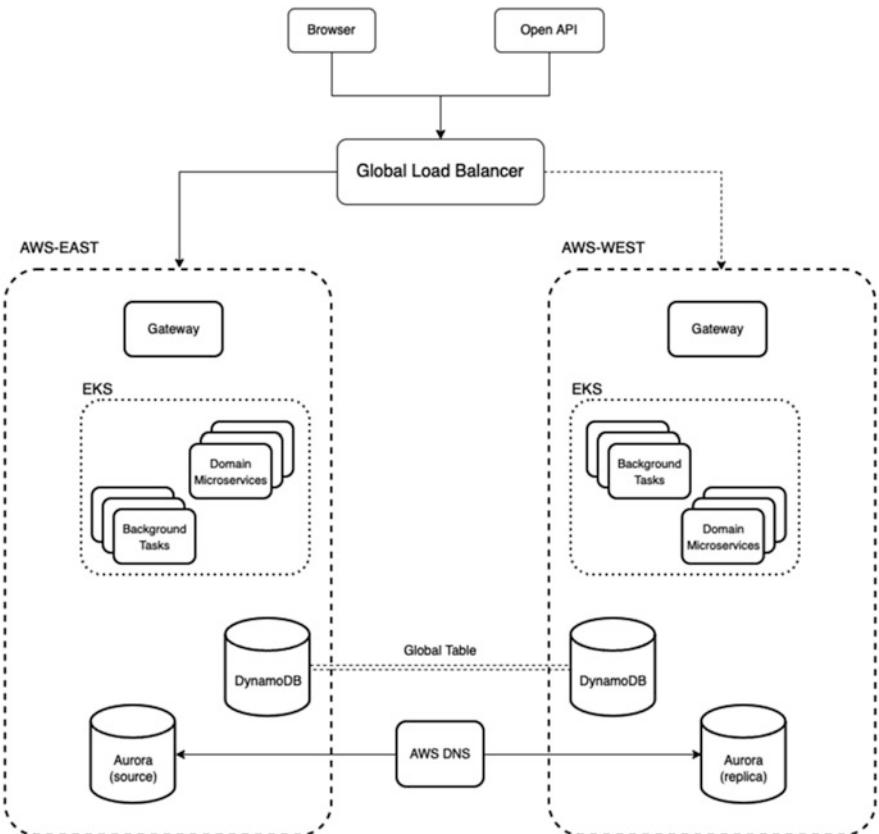


Fig. 8.34 Disaster recovery architecture overview

If the current serving data center happens to have a disaster, we need to forward the corresponding traffic to other data centers. Currently, we use AWS's GSLB service to resolve and switch the destination IP addresses of different data centers. Normally, only the IP address of the primary data center is available, and the IP addresses of other data centers are disabled.

When a disaster occurs, the health monitoring signals sent by the GSLB to the primary data center will feedback anomalies, and the GSLB will analyze the anomalies and adopt corresponding policies. If the GSLB believes that the primary data center is no longer available for service, it will set the IP address of the disaster recovery data center to be available, thus enabling rapid switchover and service recovery. The above process is shown in Fig. 8.34.

8.5 Summary

This chapter introduces our team's landing practices in terms of building a quality assurance system, where unit testing, integration testing, end-to-end testing, test automation, and chaos engineering are explored in detail, and quality assurance practices for production-like environments are also introduced.

Ensuring product quality is not just the responsibility of the testers but also of the entire team. The microservices architecture demands a higher level of collaboration effort across the entire chain from development to go-live, which means that everyone involved within the chain—developers, testers, operations engineers, etc.—needs to be accountable for what they deliver. Also, all the practices boil down to our effort into serving business value. As the business evolves, teams should reasonably adjust their quality assurance strategies and practices accordingly to build the most appropriate quality assurance system for their situation.

Chapter 9

Continuous Integration and Continuous Deployment



With the proliferation of cloud-native technologies and ecosystems, application development in an efficient manner has become the key to adopt the application to the cloud. Continuous integration and continuous deployment as accelerators have become an integral part of the development process. They are not only a product of DevOps mode for the delivery and deployment of products and software but also a collection of technologies and tools that provide powerful support for application development.

In this chapter, we will talk about the product release planning, cloud-native deployment framework, and continuous deployment support for the full lifecycle of microservice applications in the context of our team's microservice transformation practice, starting from the automation triggering, differential execution, and product archiving of continuous integration.

9.1 Git-Based Continuous Integration

In the modern world of software development, whether a service-oriented approach architecture or the agile development processes, the main objective is to increase development efficiency, and therefore application building and deployment need to keep pace with iterations.

Grady Booch mentions the concept of Continuous Integration (CI) in his book “Object-Oriented Analysis and Design with Applications” and advocates multiple integrations per day. In 2006, Martin Fowler gave a relatively formal definition of continuous integration: “Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day.” To this day, the concept of continuous integration continues to be practiced and optimized and implemented into everyday engineering development work.

After years of practice by the team, we have defined the following process for continuous integration.

- The Jenkins SubCI pipeline is automatically triggered when a software engineer makes a Pull Request from his or her individual development branch to the trunk branch of the code. The successful execution of this pipeline is a prerequisite for the code to be submitted to the trunk branch. If the execution fails, the code cannot be merged into the trunk branch.
- Once the SubCI has passed, the code is manually reviewed and merged into the trunk branch, and a post-completion pipeline (FullCI) is automatically triggered. If this pipeline is successful, the current code is working properly based on the trunk branch. Otherwise, it is necessary to find the cause of the error and fix it in time.

In order to support the above process, some preparatory work needs to be completed in advance.

- Unified configuration of Git and Jenkins to automate the triggering of the pipeline after committing and merging code.
- Write the script files required for the CI pipeline to enable code pulling, testing, packaging, and uploading.
- Create a new Jenkins task to present the CI process as a pipeline.

Of course, there may be some details to add to the specific implementation process, which we will describe later on.

The basic runtime environment for continuous integration mainly involves GitHub, Jenkins, Docker, Artifactory, etc. Building and maintaining these support tools is the cornerstone of an efficient CI implementation. Over the years, the industry has seen the emergence of new continuous integration platforms and tools such as TeamCity, Travis CI, Buddy, Drone, GoCD, CircleCI, Codeship, and many more. When building a CI infrastructure environment, we recommend choosing a technology based on your needs and the features of the tool.

9.1.1 Auto-Trigger Pipeline

In CI, we want to automate testing when merging code from different branches into the trunk branch, i.e., automatically trigger the corresponding pipeline for code testing. In practice, we generally encounter the following types of requirements in our development work.

- Once the engineer commits the code, the pipeline needs to be automatically triggered to run unit tests.
- When a service is updated, other services with which it calls or is called also need to run a test pipeline.

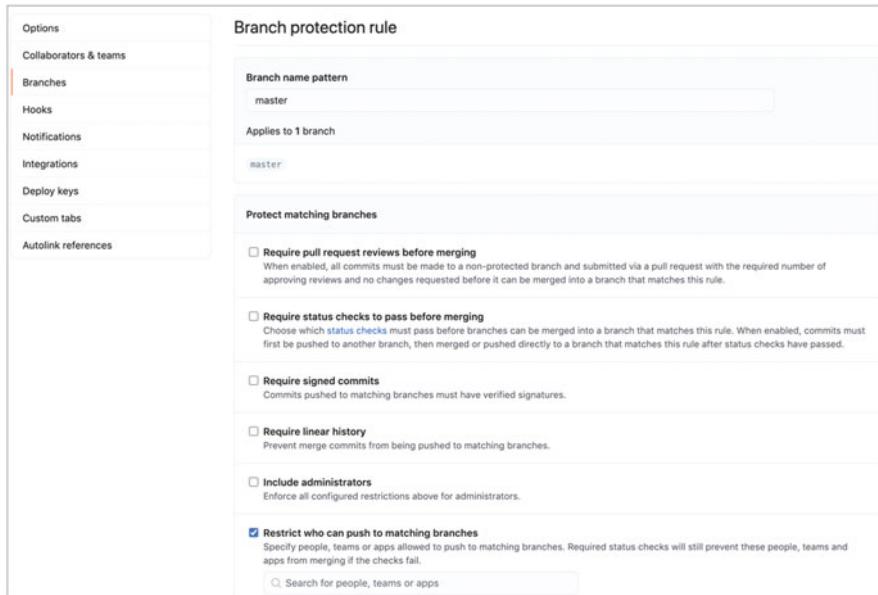


Fig. 9.1 Branch protection configuration

- Code management by repositories or organizations based on GitHub. Both types require an abstraction pipeline and different automatic triggers on the levels of the code repository or organization.
- If the changes are to test data and tool code, trigger the pipeline periodically depending on a combination of test granularity and cost.
- For microservice applications that contain multiple upstream and downstream dependencies, upstream updates need to automatically trigger the downstream pipeline.

In response to these requirements, we have identified the following solutions.

1. Automatic triggering solutions based on code repositories

The CI team implements automatic triggering based on the code repository by the these following steps.

(a) Configuring GitHub

The configuration on GitHub is as follows. First, set up the branch protection rules, as shown in Fig. 9.1, choosing the appropriate branch protection rules and making them effective on the specified branch.

Next, add the Webhook, as shown in Fig. 9.2, and set the Payload URL, for example, by filling in the URL for Jenkins to handle GitHub event requests. The events include code polling, code pushing, submission request (PR), code review, branch creation, tag creation, wiki update, and so on. CI

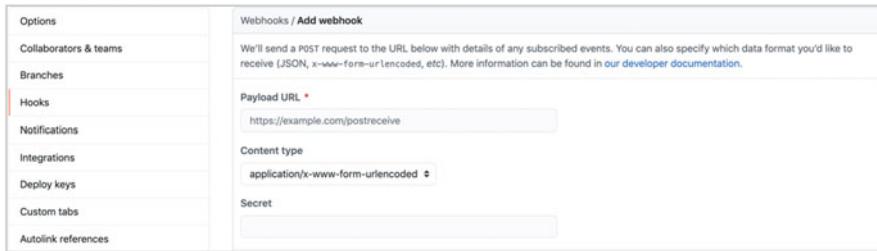


Fig. 9.2 Webhooks configuration

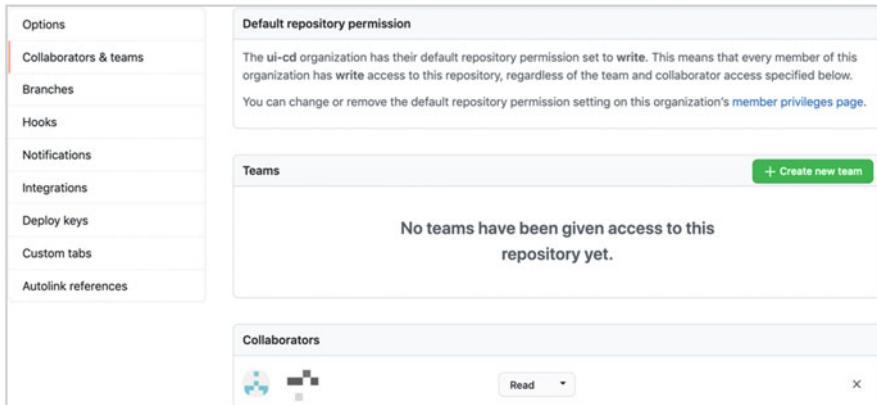


Fig. 9.3 Permission configuration

has the ability to automate triggers based on these events. We need to use a commit request as the flag event for automation triggering.

Finally, give the permission to the specific account, as shown in Fig. 9.3, to add accounts with read and write access to the code repository for Jenkins authorization configuration.

(b) Configuring Jenkins

The configuration of Jenkins involves the following steps.

First, configure the Jenkins Credential, as shown in Fig. 9.4. Generate a Credential for the account setup above with read and write permissions.

Then, configure the Jenkins MultiBranch Pipeline, such as Branch Sources configured as shown in Fig. 9.5 and Build Configuration configured as shown in Fig. 9.6.

(c) Writing Jenkinsfile

In order to automatically trigger the pipelines which have dependencies on the application logic, we build parent pipeline and sub-pipelines by Jenkinsfile. In the parent pipeline, the dependencies among these applications are recorded in a configuration file. In the sub-pipelines, the CI testing is implemented for each application. The parent pipeline triggers the



Fig. 9.4 Jenkins credential

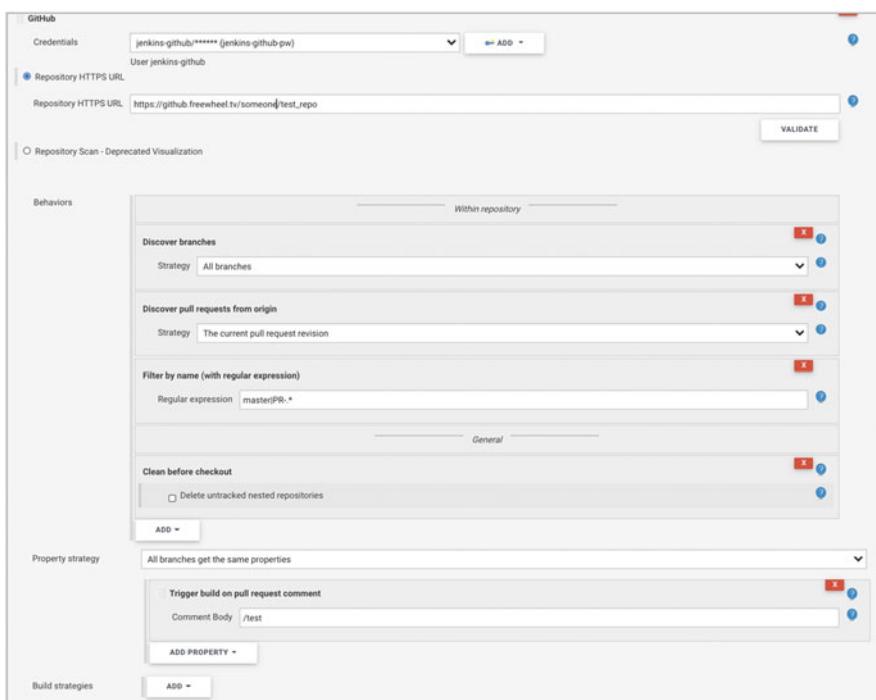


Fig. 9.5 Branch sources

Fig. 9.6 Build configuration

Build Configuration

Mode	by Jenkinsfile
Script Path	master_pipeline/uif_master_pipeline.groovy

sub-pipelines based on the dependencies in the configuration, with the configuration file MP_Configure as below.

```
ui_demo_ci_before_merge: is_pr: "true" pr_trigger_type: "merge"
owner: "owner"
include:
- src/go/src/demo_service/child_pipeline:
  - "UI/UI_CI/demo_service/demo_PR"

ui_demo_ci_after_merge: is_pr: "false"
owner: "owner"
include:
- src/go/src/demo_service/child_pipeline:
  - "UI/UI_CI/demo_service/groovy_pipeline"
  - "UI/UI_CI/demo_job/groovy_pipeline"
```

At the same time, we need to abstract out the template pipeline of multiple services, add Jenkinsfile to the code repository, and parse the services' relationship profile in Jenkinsfile, which is used to implement service updates, and automatically trigger the service's own test pipeline and other test pipelines of services with dependencies; the code example is as follows.

```
import groovy.json.

List<List<?>
> mapToList(Map map) {
    return map.collect { it -> [it.key, it.value] }
}

def slack_notification()
{}

def cleanBuildWs()
{}

def createCompileJob(Module,Job_Name,Parameters,Module_Owner) {
    // trigger subtask, process run results
}
def mpconfig_parser(MP_Configure,Change_List,Trigger_Jobs) {
    // configuration file
    parsing
}

def env_init()
{}

return this
```

Based on the above steps, the running relationship between the current pipeline and the related pipelines triggered by PR is shown in Fig. 9.7; i.e., as the parent pipeline is executed, the sub-pipelines that have an invocation relationship with the current service are also executed.

2. Automatic triggering solution based on GitHub code organization

Different approaches to code management need to be coupled with different levels of pipeline triggering solutions. The basic approach to using Jenkinsfile remains the same, but, based on the way GitHub code is organized and managed, how do you automate the creation of Jenkinsfile scripts alongside the creation of code repositories, and how do you automate the creation of MultiBranch

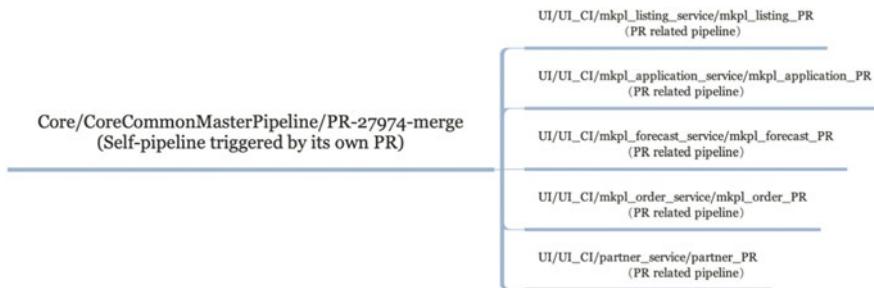


Fig. 9.7 The running relationship between pipelines

Pipelines and their configuration in Jenkins? These were all key and difficult issues in the implementation. After some research, we have identified the following solution.

(a) Automatically create Jenkinsfile scripts

First, based on the GitHub organization, we need to abstract a Jenkinsfile template that fits all code repositories and define the customization parameters that need to be passed in when creating different code repositories (the code template is shown below). When the repository is created automatically, the development team will generate a Jenkinsfile for the current repository based on the Jenkinsfile template file and the customization parameters.

```

@Library('jenkins-shared-library@main')

def Triggered_Build

def getServiceName(GIT_URL)
{}

def getSSHGitUrl(GIT_URL)
{}

pipeline{
agent {
}
environment {
}

stages {
stage('Env Init') {
}

stage("Bingo_CI_Trigger") {
// Trigger the pipeline and process the results of the run
}

post {
cleanup { cleanWs(
)}
}
}
}

```

(b) Automatic creation of MultiBranch Pipeline and its configuration parameters

After some research, the team decided to introduce the Github Organization Project feature in Jenkins. This feature automatically discovers all the repositories in a code organization, automatically configures the MultiBranch Pipeline for them and enables the automatic triggering of tasks from GitHub to Jenkins via a webhook at the organization level. The implementation also requires configuration on GitHub and Jenkins, similar to what was described earlier, and will not be repeated here.

There are two main differences between the above two solutions.

- (i) The configuration levels are different: The former is configured at the GitHub code repository level, and the latter is configured at the GitHub code organization level.
- (ii) The degree of automation is different: The former requires manual GitHub and Jenkins MultiBranch Pipeline configuration because it is a repository-level code management approach; the latter is a GitHub organization-level management approach, so after the GitHub code organization and Jenkins Github configuration is done manually, a new MultiBranch Pipeline function will be created automatically, enabling a combination of manual and automatic configuration.

In the process of solving the problem, the development team and the continuous integration team broke down organizational barriers and worked together to put DevOps into practice. It also illustrates that continuous integration requires not only meeting development needs at the code and tool level but also the active involvement of the development team, with the developers providing constructive suggestions on detailed implementation.

3. Other solutions

For periodic trigger requirements, our main solution is to customize the trigger conditions via the periodic trigger feature on Jenkins, as shown in Fig. 9.8 under the “Build periodically” option.

For relationship-based triggering requirements, we need to understand the main differences between upstream and downstream relationship-based triggering pipelines and parent-child relationship-based triggering pipelines in microservices: the success or failure of the child pipeline affects the execution results of the parent pipeline, whereas the focus of the upstream and downstream pipelines is only on the backward and forward connection, and the execution results of the downstream pipeline do not affect the execution of the upstream pipeline; in addition, parent-child pipeline is implemented in a scripts way that facilitates version control and retrieval, whereas upstream and downstream pipelines are usually configured directly in the Jenkins GUI, as shown in Fig. 9.8 under the “Build after other projects are built” option. Of course, the choice of whether to configure the Jenkins pipeline through the GUI or through code should be analyzed and chosen according to your specific needs.

Fig. 9.8 Build triggers

Finally, as you can see in Fig. 9.8, Jenkins offers a wide variety of pipeline triggers, and we need to be flexible in choosing them in relation to our actual needs, not just for the sake of using them but following the principle of making the tools serve the core business.

9.1.2 *Pipeline Differentiation and Unified Collaboration*

Section 9.1.1 describes how the pipeline is triggered. This section will present the functional perspective of the continuous integration pipeline in terms of its differentiation and unified collaboration across all phases of software development, based on our practice.

1. Pipeline differentiation

Firstly, from the point of view of code merging and the function of the pipeline itself, pipelines can be divided into a global pipeline (Main pipeline), a pre-pipeline (SubCI pipeline), and a post-pipeline (FullCI pipeline). Pre and post are relative to the event that the code is merged into the main branch.

The function of the global pipeline is to determine whether the next step to be executed is SubCI or FullCI based on the branch the code currently belongs to; if the branch it currently belongs to is not a trunk branch, SubCI is executed, as shown in Fig. 9.9.

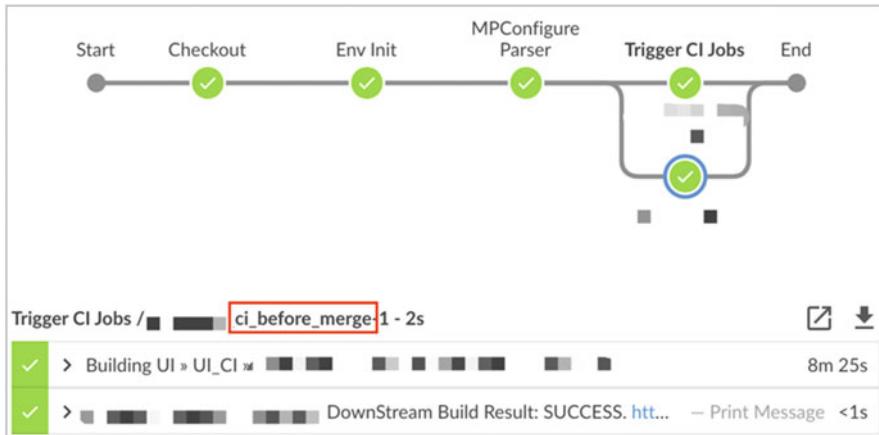


Fig. 9.9 SubCI pipeline

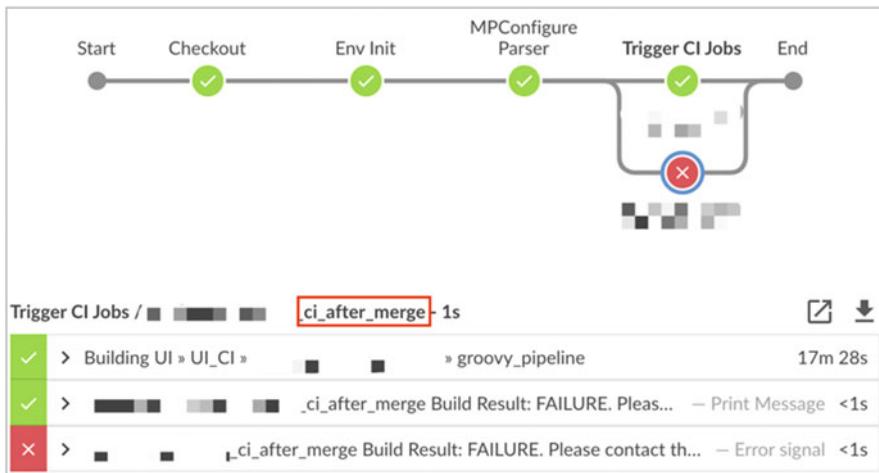


Fig. 9.10 FullCI pipeline

If the branch you currently belong to is a trunk branch, execute FullCI, as shown in Fig. 9.10.

As can be seen from the naming, the main purpose of SubCI is to perform the necessary tests to ensure the quality of code changes before they are merged into the trunk branch. Therefore, it is usually a subset of FullCI and typically includes functions such as code pulling, unit testing, and code compilation verification.

FullCI is more feature-rich, with code pulling, unit testing, functional testing, regression testing, integration testing, and smoke testing. It is also responsible for the generation and upload of deployment packages, as shown in Fig. 9.11. The deployment packages are described in detail in later sections.



Fig. 9.11 Functions in the FullCI

Of course, there are customized features that can be integrated into the CI pipeline depending on the requirements. As long as they follow CI principles and are in line with DevOps thinking, we are willing to experiment with an open mind.

Secondly, in terms of code type, pipelines can be divided into business logic code pipelines and auxiliary tool type code pipelines. Business logic code pipelines are of high importance and need to be created as a priority. Auxiliary tool code can be very broad in scope and can include code for testing tools, managing application data, or maintaining the environment. These code pipelines have different functions depending on what they correspond to.

As the application continues to improve, so does the pipeline for these ancillary tool-like codes. For example, the CI pipeline for the mock testing tool is mainly used for testing the mock tool, compiling it and uploading and archiving the container image. In the business pipeline, updates to the program execution environment can also be controlled through the CI process. For example, the CI pipeline for the test environment can complete the construction of the test environment based on Docker Compose as well as additional test data and update the regression test database synchronously. Also, we create a PostCheck pipeline to execute application-level test cases and send test results to the person in charge via email and instant messaging. In the process of applying to the cloud, we built a CI pipeline for AWS AMI images and a CI pipeline for data maintenance of the regression test database based on Packer, which periodically completes the task of updating test data.

2. Unified collaboration in the pipelines

By triggering a pipeline at a fixed point in time and at a fixed frequency, it is possible to ensure that the products of this pipeline are applied to other pipelines periodically and on-demand, which is a manifestation of unified collaboration.

Formally, both parent-sub relationship pipelines and upstream-downstream relationship pipelines are a manifestation of unified collaboration. Alternatively, unified collaboration relationships exist between pipelines based on multiple branches, as shown in Fig. 9.12.

- (a) When a developer creates a PR based on a branch, GitHub sends a webhook with the PR information to Jenkins.
- (b) Jenkins receives the PR and then automatically creates a pipeline for that branch, which uses the created pipeline's Jenkinsfile to perform the tasks for each stage. At this point, the PR merge will be blocked until Jenkins returns to the build state.

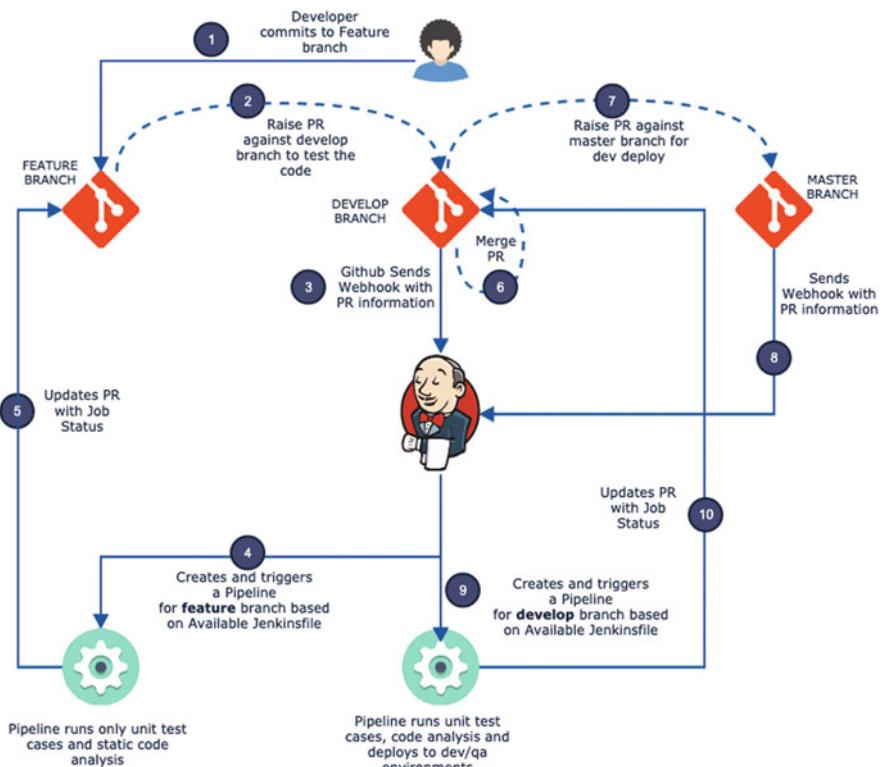


Fig. 9.12 (<https://devopscube.com/jenkins-multibranch-pipeline-tutorial>)

- (c) Once the build is complete, Jenkins will update the PR status on GitHub. If the pipeline executes successfully, the code is ready to be merged. If you want to check the Jenkins build log, you can find the Jenkins build log link in the PR status.

The pipeline runs within the Jenkins Workspace, and there is a unified collaboration between the multiple components of the different testing stages.

- (a) In the unit testing stage, it tests how well the data artifacts of the data pipeline support the unit test.
- (b) In regression testing, it is required to build a test environment with minimal dependencies for the service under test. This type of test environment contains at least several microservices and data products from the test data pipeline and may also contain tool products from the mock test tool pipeline, which also need to work in unison with each other.

Looking at the CI environment as a whole, from the AMI in EC2 to the Docker-Runner image to the various components in the Jenkins workspace,

there are products of the CI pipeline at different levels, which is an indication of the unified collaboration of the pipeline.

9.1.3 Pipeline Product Storage Planning

The static products output from the different pipelines not only provide data for quality assurance but also traceable deployment packages for product releases. Therefore, we need to plan the storage of the pipeline products.

1. Pipeline products

Pipeline products can generally be divided into three categories.

- (a) Quality assurance data
- (b) Auxiliary tools products
- (c) Deployment files (including images and packages)

The main products generated from application-related pipelines are code static analysis data, unit, functional and regression test coverage data, test log files, application binaries, deployment packages, etc. The products generated from the auxiliary development pipeline are test datasets and test tools.

In our practice, applications are made in binary format and built into Docker images, which are then distributed via Helm Chart. The Dockerfile is shown as below.

```
FROM alpine:3.5

RUN mkdir -p /opt/demo/tlog && \
adduser demo

USER demo

WORKDIR /opt/demo/current
COPY ./bin /opt/demo/current/bin

CMD ["/bin/sh","-c","/opt/demo/current/bin/demo_service start -c /opt/demo/current/env-config/demo.yml -G true"]
```

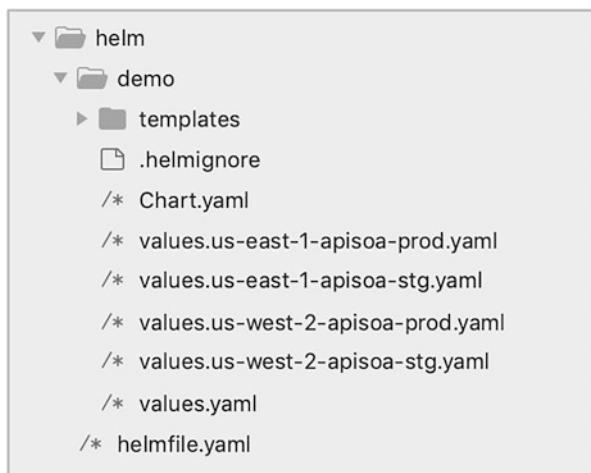
The Helm Chart file directory structure is shown in Fig. 9.13.

2. Product storage planning

Quality assurance data is presented after the pipeline that produced it has been executed. Artifactory is a repository for pipeline products, which is essentially a server-side software for Maven repositories, supporting multiple development languages, metadata retrieval in any dimension, cross-language forward and reverse dependency analysis, deep recursion, offsite disaster recovery, and other Enterprise level features.

Each service within a microservice application is deployed in a uniform way, with a corresponding Docker image and Helm Chart deployment package. With each update of the code, the version of the Docker image and Helm Chart

Fig. 9.13 Helm chart structure



deployment package will also change, so we need to follow these key principles when planning our product storage:

- (a) Deployment of product sorting storage.
- (b) Deployment products are uniquely identified.
- (c) The product of the deployment is traceable.
- (d) Permission control when deploying product uploads.
- (e) Deployment products are regularly cleaned.

The Artifactory saves the products in the repository according to their file types, mainly in the image category, the deployment Helm package category, the npm category, etc. Products from the same pipeline are also saved under different categories of paths, as shown in Fig. 9.14.

Services are distinguished from package versions by a unique identity, and there will be multiple package files with different package versions in the same service, as shown in Fig. 9.15. The naming rule of versions is: service component

Fig. 9.14 Artifact repository

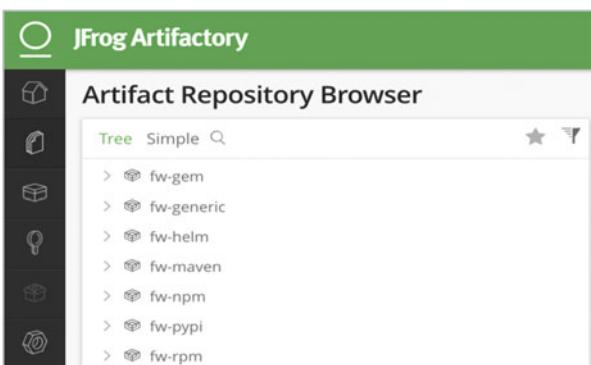
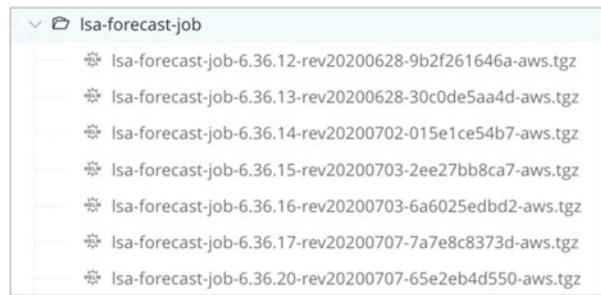


Fig. 9.15 Multiple packages



name—main version number, sub-version number, Pipeline build number—date generated—current code commit hash abbreviation—AWS tag', e.g., demo-services-6.36.12-rev20200628-9b2f261646a-aws.tgz.

By uniquely identifying, we can find out which build task in which pipeline the product originated from and thus determine which version of the source code corresponds to that version, for the purpose of tracing and locating the code.

We also control the permissions for product uploads to ensure that valid products can be uploaded to the Artifactory repository, blocking invalid uploads, and also rationalize the use of the Artifactory repository to avoid overuse of repository resources. In addition, files in the repository need to be cleaned regularly. However, it is important to note that the cleanup is of pipeline products and not of the source code or scripts that generate them. This means that the ability to recreate the product from the source code or script after it has been deleted is a prerequisite for safe cleanup.

9.2 Helm-Based Continuous Deployment

CI is a prerequisite for CD and CD is the purpose of CI. In general, the development steps related to CI/CD are as follows.

- Once the developers have implemented the functionality and completed local testing, the CI pipeline is used to complete a full-quality test and generate a deployment package.
- In accordance with the release plan, developers use the CD pipeline to release deployment packages to pre-release and production environments and to perform functional validation.

To support the above processes, the following priorities need to be considered when implementing continuous deployment.

- Select deployment tools based on system architecture, deployment environment, and release time.

- Write the script files required for the CD pipeline, covering the acquisition of deployment configurations, Helm release versions, necessary tests after deployment, etc.
- Create a new Jenkins Job to present the CD process as a Job.

In the process of service adoption cloud platform, our deployment environment migrated from physical machines to Kubernetes-based cloud servers, and the deployment packages changed from RPM/TAR packages to Helm packages, which required the scripts in the CD pipeline to be updated to meet the requirements of the cloud platform adoption.

The base runtime environment for continuous deployment mainly involves Kubernetes clusters, Jenkins, Docker, Artifactory, etc. They are not only the dependencies of CD, but, more importantly, the stability of these services is directly related to whether the service-level agreement (SLA) for the production environment can be reached. This section will focus on the deployment planning and microservices release to the cloud in the CD process.

9.2.1 Deployment Planning

To implement the continuous deployment, you need to analyze the three main aspects of the application—system architecture, deployment environment, and release time—and then consider the selection and planning of deployment tools.

1. System architecture

In a microservice application, each service implements a different business, but the compilation and product form is the same; they all package the compiled binaries and configuration into a Docker image. This allows each service to be deployed independently, and this unified application form also facilitates uniform deployment steps.

If deploying an application based on a Kubernetes cluster, the preferred deployment tools are Helm or Kustomize, and, given the need to manage the application lifecycle and versioning, we chose Helm, the Kubernetes package manager, as the deployment tool. Therefore, we added the Helm Chart directory to the code structure of each service, completed the packaging and uploading of the Helm package in the CI pipeline, and used the package to publish the application in the CD pipeline.

2. Deployment environment

Currently, our main deployment environments are the pre-release and production environments. They have three separate clusters, as shown in Fig. 9.16.

- (a) EKS-based Eastern US Cluster (AWS-EAST:EKS).
- (b) EKS-based US West Cluster (AWS-WEST:EKS), which serves as a disaster recovery cluster interchangeably with the East Cluster.

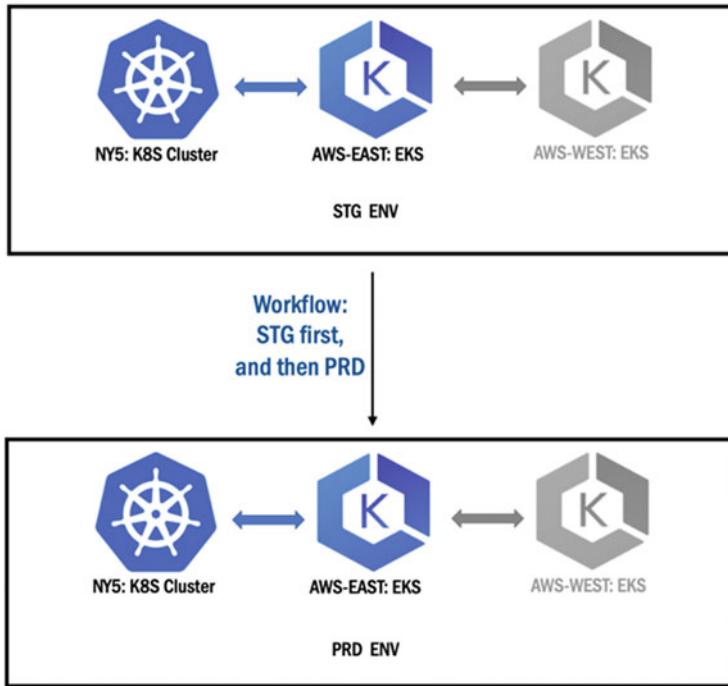


Fig. 9.16 Deployment environments

- (c) The original Kubernetes cluster (NY5:K8S Cluster), which was built on physical machines, is being deprecated as planned.

The microservices contained in the application are usually deployed centrally within the same namespace (namespace) in the cluster for ease of call between services and maintenance. In other namespaces, infrastructure components such as Service Mesh, monitoring, etc. are also deployed.

For the above deployment environments, the pipeline needs to have the ability to publish applications to different environments, as well as the ability to deploy applications to different clusters within the same environment. In addition, because there are dozens of microservices, the logical framework for the deployment steps needs to be highly abstracted and templated. This will meet the needs of different services, different environments, and different clusters.

3. Release time

The release cycle for an application is typically 4 weeks. The first 2 weeks are spent developing, after which the release is deployed to a pre-release environment, the next 2 weeks are spent running tests, and the fourth week is spent releasing the release to a production environment.

In addition, if bugs are found online that affect users and need to be fixed in a timely manner, they can be deployed at any time after risk assessment and

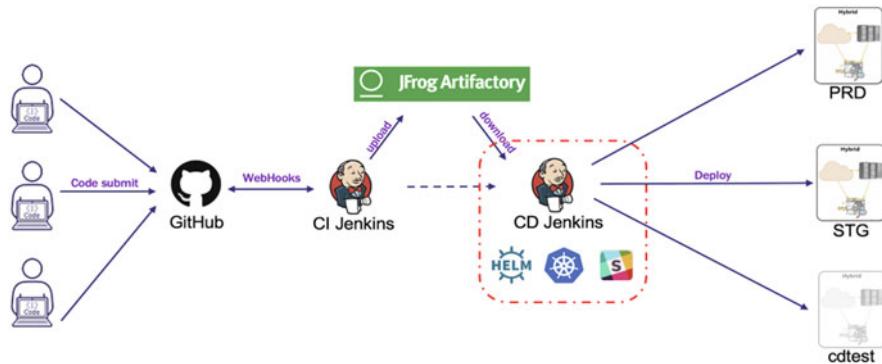


Fig. 9.17 Overall development workflow

approval, rather than following a set release cycle. To be on the safe side, deployments still need to be made to a pre-release environment before being deployed to production.

9.2.2 Framework for the Multiple Clusters Deployment in Different Environments

Applications need to be deployed to multiple Kubernetes clusters in different environments. Deployment packages are generated in the CI pipeline. First, the developer commits the code to GitHub, which automatically triggers the CI pipeline; the deployment package is generated in the pipeline and uploaded to the Artifactory, as described in Sect. 9.1. The CD pipeline then downloads the deployment package from the Artifactory and deploys it to a cluster of different environments. The overall flow is shown in Fig. 9.17.

The continuous deployment in Fig. 9.17 is executed using Jenkins jobs, and the deployment framework is based on Helm, which also automatically performs Pod-level and service-functional-level validation after deployment.

1. Introduction to Helm

Helm is a package management tool for Kubernetes applications; we define, install, and upgrade complex Kubernetes applications mainly through Helm Chart. Architecturally, it is a C/S architecture consisting of a client and server; the server is integrated into a Kubernetes cluster called Tiller, which has been deprecated after Helm 3.

There are three other important concepts in Helm: Chart, Release, and Repository. The Helm client gets the Chart from the Repository and passes the deployment file to Tiller for processing, generating a list of Kubernetes resources, and Tiller interacts with the Kubernetes API server to request resources and install the

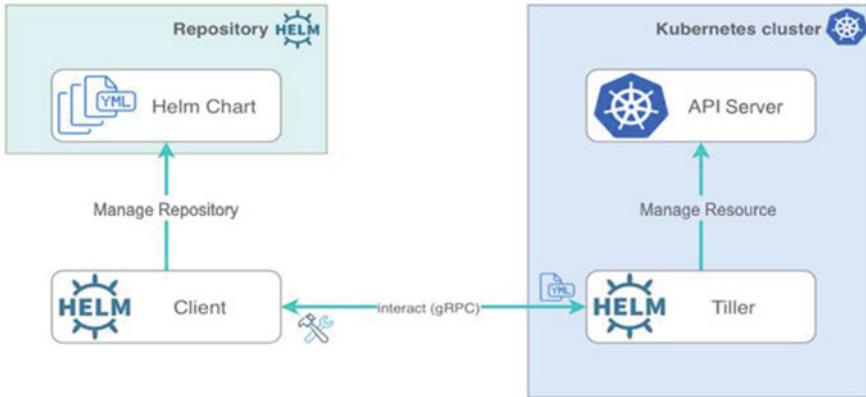


Fig. 9.18 Helm architecture

application to the Kubernetes cluster. During this process, the Chart instance running on the Kubernetes cluster is the Release as defined by Helm. The flow of interaction between Helm and the Kubernetes cluster is shown in Fig. 9.18.

2. Dividing Jenkins tasks by environment and cluster

Service deployment tasks are for development, pre-release, and production environments. We have therefore divided these Jenkins tasks into three corresponding folders, ESC-DEV, STG, and PRD, for the purposes of categorization management and deployment misuse prevention. Within the folder corresponding to each environment, there are four subfolders. Take the pre-release environment folder STG, for example, the four subfolders are STG-AWS, STG-AWS-WEST, STG-NY5, and STG_Both_NY5_AWS (including both NY5 and AWS environments), as shown in Fig. 9.19.

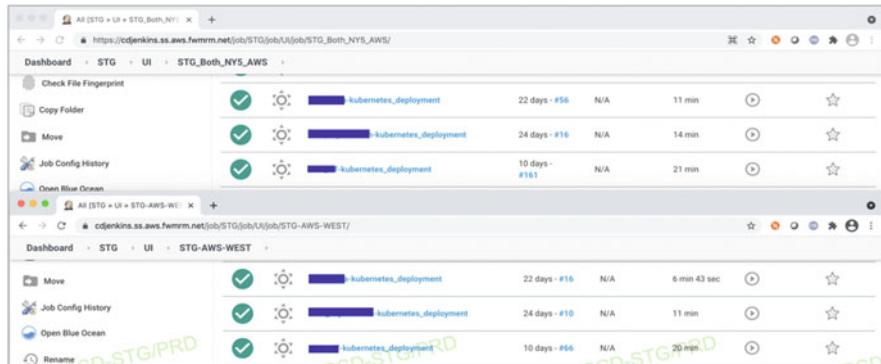
The STG_Both_NY5_AWS subfolder holds the deployment parent tasks, which are used to trigger the subtasks in the three clusters deployed to the current environment to execute them in parallel. The other three subfolders hold the deployment subtasks for the different clusters in the current environment, executing the deployment tasks in the AWS-EAST-EKS, AWS-WEST-EKS, and NY5-K8S Cluster clusters.

For example, if the STG_Both_NY5_AWS folder has a parent task like demo-kubernetes-deployment, then the other three subfolders have "demo-kubernetes-deployment" subtasks with the same name. When deploying the demo service in a pre-release environment, simply execute the demo-kubernetes-deployment parent task in the STG_Both_NY5_AWS folder to automatically trigger the child task, which will then perform the actual deployment operation, enabling deployments in multiple different clusters to be completed with a single parent task.

Using the deployment tasks in the STG_Both_NY5_AWS folder and the STG-AWS-WEST folder as an example, Fig. 9.20 shows a list of child tasks of a parent task.

S	W	Name ↓	Last Success	Last Failure	Last Duration	Built On	Fav
		ECS-DEV	N/A	N/A	N/A	on STG/PRD	★
		PRD	N/A	N/A	N/A		★
		STG	N/A	N/A	N/A		★

S	W	Name ↓	Last Success	Last Failure	Last Duration	Built On	Fav
		STG-AWS	N/A	N/A	N/A	AWS	★
		STG-AWS-WEST	N/A	N/A	N/A		★
		STG-NYS	N/A	N/A	N/A		★
		STG_Both_NYS_AWS	N/A	N/A	N/A		★

Fig. 9.19 Folders for Jenkins tasks**Fig. 9.20** Jenkins tasks in folders

Of course, parent and child tasks are also supported for manual triggering and execution, and Fig. 9.21 shows the configuration interface for parent and child tasks.

On the left side is the parent task configuration screen, with optional parameters such as target cluster, deployment package name, approval tickets, and deployment code branch. The main function of the parent task is to trigger the specified subtasks based on the parameters and to pass these parameters to the subtasks.

On the right side is the subtask configuration screen with optional parameters such as target cluster, deployment package name, approval tickets, deployment steps, and deployment code branches. The focus of the subtasks is on reading the configuration file based on the parameters, executing the Helm deployment

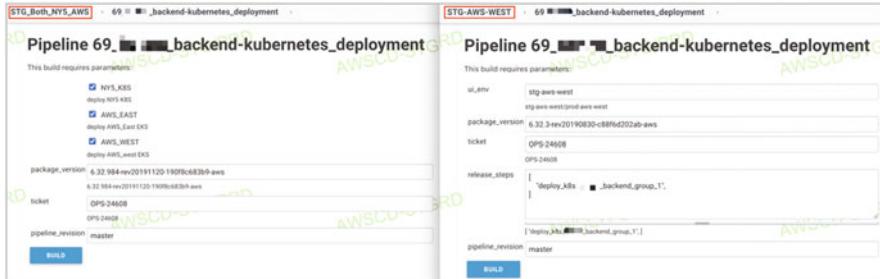


Fig. 9.21 Jenkins jobs parameters

commands and basic validation of the deployment once it is complete, which is the core of the deployment framework. The tasks used for service deployment are basic Jenkins tasks with parameters that can be added on demand, and the main functionality is implemented in Groovy files. It is important to note that simultaneous execution of deployment tasks is not allowed, to ensure that the deployment is executed serially.

3. Deployment framework

The deployment framework performs deployment and validation operations based on the configuration of the microservice; the structure of which is shown in Fig. 9.22. The service deployment subtask executes the Groovy file of the deployment portal directly, parses the parameters of the task in the generic deployment unit and invokes a Python script, then executes the Helm command in Python to deploy based on the service configuration file, and then invokes the Kubectl command for Pod-level deployment validation.

Firstly, each service corresponds to a configuration file, which clearly lists the deployment operations and basic validation points to be performed in different clusters in different environments, as follows. If you want to support the deployment of new services, you need to add the configuration file for the new service.

```
cdtest-ny5:
...
...
... cdtest-aws-east:
...
...
... cdtest-aws-west:
...
...
...
stg-ny5:
...
...
... stg-aws-east: deploy_steps: deploy_k8s_demo postcheck: check_pod_image_version: demo check_pod_number: demo!1 check_pod_status: demo!
Running stg-aws-west:
...
...
prod-ny5:
...
...
... prod-aws-east:
...
...
... prod-aws-west:
...
...
...
```

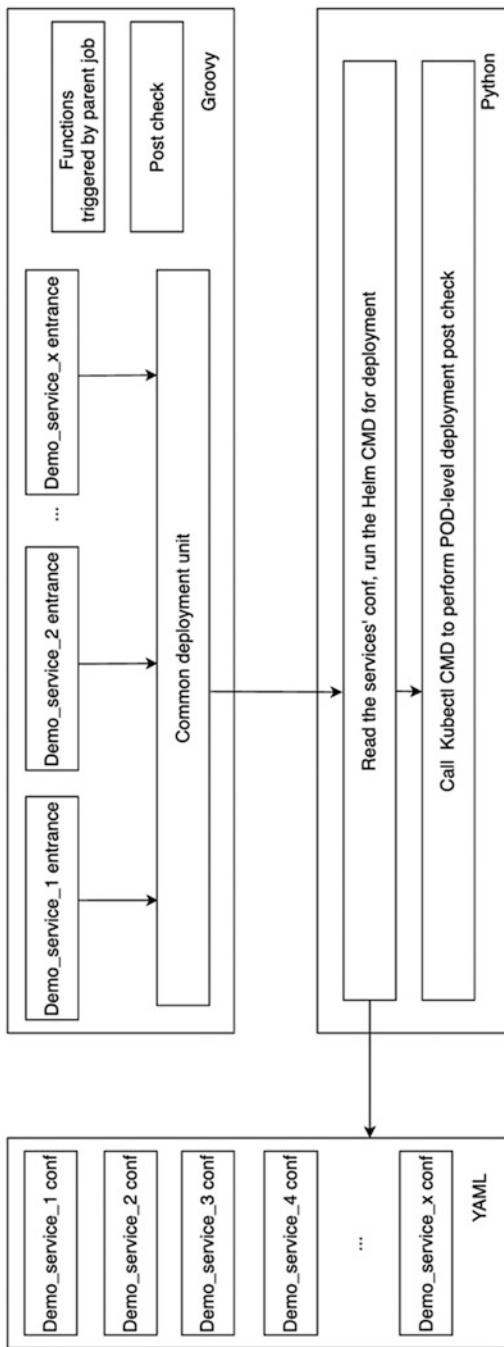


Fig. 9.22 The deployment framework

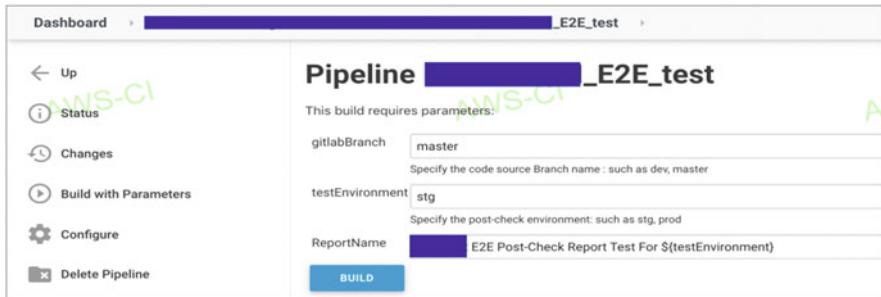


Fig. 9.23 A test subtask interface

Then, after the configuration file is parsed and the Jenkins task parameters are determined, the service can be deployed via the Helm command, with the following code:

```
def deploy_k8s_soa_service_package(service,env,version,jira_result)
:### Service deployment via Helm command helm_install_command = "helm upgrade " + Helm_release_name + chart_version + " --install --wait --
timeout 1200" (status,output) = execute_command_shell(helm_install_command,jira_result)
...
...
return ret,jira_result
```

Finally, after the Helm has been successfully deployed, the Pod-level deployment is verified using the Kubectl command with the following code:

```
def check_pod_image_version(**dictArgs): (status,output) = execute_command_shell("kubectl describe pod " + pod_name + " ." + version + " | grep -oP
\"Image:\$+[\\w\\.-]+[\\w\\.\\/]+[\\w\\.-]+[\\w\\.-]+[\\w\\.-]+\"|grep " + pod_name + "|cut -d: -f3 ",jira_result)
...
...
return ret

def check_pod_number(**dictArgs): (status,output) = execute_command_shell("kubectl get pod -a -o wide | grep -P ^" + pod_name + "-" + version,jira_
result)
...
...
return ret

def check_pod_status(**dictArgs): (status,output) = execute_command_shell("kubectl get pod -a -o wide | grep -P ^" + pod_name + "-" + version +
 "|awk '{ print \$3}'",jira_result)
...
...
return ret
```

4. Service functional level validation after deployment

During the execution of a Jenkins parent task, in addition to triggering sub-tasks for actual deployment, you can also trigger service-level functional validation. These validation operations are also Jenkins subtasks, and Fig. 9.23 shows a test-related subtask interface.

The verification is of the test cases for the service function. If a test case fails to execute, a notification can be made via email or instant messaging; Fig. 9.24 shows an example of using email to notify a task failure.



Fig. 9.24 A notification email

9.2.3 Cloud-Native Support and Task Maintenance

The Jenkins tools used for continuous integration and continuous deployment are all deployed on AWS EC2 (Elastic Compute Cloud), with AMI (Amazon Machine Images) and Docker images generated by scripts that can be reused when acquired. Currently, the deployment of multiple clusters in different environments of microservices requires the support of over two hundred Jenkins tasks; the maintenance of which we also automate.

1. Jenkins support for cloud-native

The Jenkins node performing the continuous deployment uses EC2 provided by AWS and requires support for Helm, Kubectl, Slack, and Jira, which are less demanding on system resources such as CPU, etc. The AMI running on EC2 is based on the Packer tool and is generated through a pipeline execution script. The creation of the pipeline is similar to that described previously, the difference lies in the Groovy file for execution and the packer.json file, and the contents of the Groovy file are as follows:

```

pipeline {
    agent {
        label 'ui-server-base'
    }
    options {
        environment
        AMI_Name = "${ami_name_prefix}" + "." + "${ami_name_postfix}"
    }
    stages {
        stage('Validate Packer File'){
            //Validate packer.json for validity
        }
        stage('Build AMI'){
            ...
            // use Packer for AMI production
            ...
            /packer build -var
            \${AMI_Name}=\${AMI_Name}\${ami_name_prefix}/packer.json
            ...
        }
    }
}

```

The packer.json file is as follows and is mainly used to describe the toolset that needs to be installed in the built AMI and the AWS environment that Packer needs to run.

```
{
  "_comment": "Used by xxxxxx label",
  "variables": {"AMI_Name": "xxxxxx"
  },
  "builders": [{"ami_name": "{{user 'AMI_Name'}}", "ami_users": ["xxxxxx", "xxxxxxxx"], ...
  ...
  ...
  },
  {
    "tags": {
      ...
      ...
      ...
    },
    "run_tags": {
      ...
      ...
      ...
    },
    "ami_block_device_mappings": [
      {
        ...
        ...
        ...
      }
    ],
    "source_ami": "xxxxxxxx",
    "region": "us-east-1",
    ...
    ...
  }],
  "provisioners": [
    {
      "type": "shell",
      "inline": [
        ## install kubectl",
        ## install helm 2.13.1",
        ## install jira client",
        ...
        ...
      ]
    }
  ]
}
```

After execution of the deployment, the Jenkins node for service functional level validation needs to support the execution of the test cases, so a basic test environment has to be set up on the Jenkins node. The approach we took was to build the test cases and execution environment as a Docker image, use EC2 as the host, and run the Docker image on the host for verification purposes. The creation of the pipeline is similar to that described previously, the difference lies in the Groovy file and the Dockerfile file, and the contents of the Groovy file are as follows:

```
pipeline {
  agent {
    docker {
      image 'arti.freewheel.tv/xxxxx/demo:latest'
      label 'demo-server-post-check'
      args '-u root xxxxxx xxxxxx --network="host"'
    }
  }
  ...
}
```

The contents of the Dockerfile file are as follows and are used to complete the installation and configuration of the test case dependency environment:

```
FROM centos:  
centos# Install and configure the required packages.  
...  
...  
RUN /bin/bash -l -c "nvm install 12.16.3 && npm install -g newman yarn"  
CMD ["/bin/bash", "-D"]
```

2. Maintenance of Jenkins tasks

Daily maintenance of Jenkins tasks includes creation, update, deletion, and alarm handling. When a new service goes online, the task needs to be created; when the service goes offline, the corresponding task needs to be deleted. Maintenance is mainly about batch updating of task parameters and alarm handling. Batch updates are implemented via the Jenkins API in conjunction with a Python multi-threaded script, with the following code:

```
def update_config(base_job_name):  
...  
...  
    job = server.get_job(full_job_name)  
    config = job.get_config()  
    new_config = config.replace('old_para', 'new_para')  
    job.update_config(new_config)  
    ...  
  
if __name__ == '__main__':  
    jenkins_host = 'https://cdjenkins.demo.net/'  
    base_job_names = [  
        'demo-kubernetes_deployment',  
        ...  
    ]  
    ...  
    ...  
    server = Jenkins(jenkins_host, username = 'XXXXXXXXXX', password = 'XXXXXXXXXX')  
    for base_job_name in base_job_names:  
        job_para_update = MyThread(update_config, args=(base_job_name, ))  
        job_para_update.start()  
        threads.append(job_para_update)  
    for i in threads:  
        i.join()  
  
    for i in threads:  
        ret += i.get_result()  
  
    print("ret = %d" % (ret))
```

We handle alarms by manually analyzing the Jenkins task logs. Firstly, we maintain Jenkins tasks with real-time alerting capabilities. As soon as a task goes wrong, Jenkins sends an instant message to the engineer executing the task multiple times with information about the deployment environment and version of the alarm in the message until the task is processed. The engineer can follow the alarm link to jump directly to the Jenkins task and then analyze the error logs to diagnose the root cause of the alarm. If it is an error in the parameters of the Jenkins task, the current task needs to be stopped and retriggered; if it is a problem with Jenkins itself (e.g., unable to assign the corresponding Jenkins server), the Jenkins service needs to be troubleshooted; if it is a failure of the Helm

deployment in the Jenkins task, the failure needs to continue to be analyzed according to the prompts.

9.3 Continuous Deployment Practices Based on Kubernetes

We implement and run microservice applications based on Kubernetes clusters, deployed at service granularity. To better serve customer traffic and reduce deployment costs, each service's Helm Chart uses automatic horizontal scaling on top of Kubernetes resource quotas. In addition, we need to follow a regular process from the time the service is first brought online, upgraded, and then finally taken offline. If you encounter a deployment failure, you need to analyze the specific problem.

9.3.1 *Pod Resource Quotas and Horizontal Autoscaling*

Multiple microservices typically share cluster resources with a fixed number of nodes. To ensure that each service is allocated the right amount of resources, we use the resource quota tool provided by Kubernetes.

Resource quotas are defined through the `ResourceQuota` object and are limited by the total amount of resources consumed in the namespace. The total number of objects of a certain type can be limited to an upper limit in the namespace, as well as the upper limit of computing resources used by Pods. For example, when quotas are enabled for computing resources such as CPU and memory, each service must set a request and a limit for these resources; otherwise, the Pod will not be created successfully because of the quota.

We use Helm Chart's `deployment.yaml` and `values.yaml` to implement the deployment of the application; examples of which are shown below:

```
# Specify the resources restricted to the container in deployment.yaml
apiVersion:
apps/v1kind:
Deploymentmetadata:
...
...
spec:
strategy:
type: RollingUpdate rollingUpdate: maxUnavailable: 0 maxSurge: 3
replicas: {{ .Values.replicaCount }}
...
...
template:
metadata:
...
...
spec: serviceAccountName: {{ $name }}
volumes:
...
...
containers:
- name: {{ .Chart.Name }}
image: ...
...   livenessProbe:
...   readinessProbe:
...   volumeMounts:
...
...
resources:
# References the contents of the values.yaml file
{{- toYaml .Values.resources | nindent 10 }}
...
...
{{- end }}

# Specify in values.yaml the specific values that the container is limited to
replicaCount: 3
...
...
resources:
limits:
cpu: 1
memory: 1Gi
requests: cpu: 10m
memory: 128Mi
```

At certain times of the day, there is a dramatic increase in the traffic accessing the application. In order to improve the availability and scalability of the service while reducing the cost of deploying the application, we use the Horizontal Pod Autoscaler (HPA) feature. It automatically scales the number of Pods based on CPU utilization or memory usage. In addition to this, HPA can also perform automatic scaling based on custom metrics. Of course, if some objects in Kubernetes (e.g., DaemonSet) do not support scaling themselves, HPA is not applicable.

Pod automatic horizontal expansion and contraction are achieved by defining the `HorizontalPodAutoscaler` object and adding it to the Helm Chart, as shown in the example below:

```

apiVersion:
autoscaling/demokind:
HorizontalPodAutoscalermetadata:
name: demo
labels:
...
...
spec: maxReplicas: 20 minReplicas: 6 scaleTargetRef: apiVersion: apps/v1
kind: Deployment
name: demo
metrics:
- type: Resource resource:
  name: cpu
  target:
    type: AverageValue averageValue: {{ $.Values.features.hpa.cpuAverageValue }}
- type: Resource
  resource: name
  :memory target
  :type
  :AverageValue averageValue: {{ $.Values.features.hpa.memoryAverageValue }}
...
...

```

When using Helm Chart for deployment upgrades, you can view the log file of the HPA in effect, as shown in Fig. 9.25.

By combining Pod resource quotas with horizontal scaling, we achieve a rational use of cluster resources while being able to cope with different orders of magnitude of access requirements.

9.3.2 Service Online/Offline Workflow and Fault Analysis

During the continuous deployment of applications, we address issues that arise during the use of Helm and Kubernetes clusters, such as network connectivity issues, access permissions issues, and resource limitations. Depending on the specific deployment scenario, we will introduce how to perform fault analysis from three aspects: service online process, service version upgrade, and service offline process.

1. Failure analysis of the service online process

In conjunction with the previous introduction, the service online process is to test the service functionality and Helm Chart in a development environment, followed by deployment testing and system-level User Acceptance Test (UAT) in a pre-release environment, and finally a formal online release.

Although we can use various strategies for testing, the first deployment in a production environment is the real test. What needs to be noted when a service goes live is whether its dependencies are already in the cluster. As the complexity

demo/HorizontalPodAutoscaler						
NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
demo-api-grpc	Deployment/demo-api-grpc	95m/800m, 124950118400m/500Mi	6	20	7	259d
demo-api-http	Deployment/demo-api-http	39m/800m, 319315512888m/500Mi	6	20	8	259d

Fig. 9.25 HPA information



```
helm upgrade
--install --wait --timeout 1200
UPGRADE FAILED
ROLLING BACK
Error: timed out waiting for the condition
Error: UPGRADE FAILED: timed out waiting for the condition
```

Fig. 9.26 Helm error message

of the application increases, so does the number of dependencies on the service itself. Some dependencies are between services, and some dependencies are external to the service, such as the MySQL database, Solr search server, Amazon S3 (Simple Storage Service) storage service, etc.

Currently, our automated deployment tasks support the display and notification of deployment results and Pod information. However, when there is a problem with the deployment, manual intervention is required to analyze it. In conjunction with the logging information in the Jenkins deployment task, we used the Kubectl command to troubleshoot the problem, and the analysis steps are as follows.

First, check the log in the deployment task.

If the error message as shown in Fig. 9.26 appears, it indicates that the Helm Chart deployment failed within the set timeout of the 1200s.

Pod-level troubleshooting is then performed, which is viewed via the command line as follows:

```
$ kubectl get pods -n ui-app | grep
demodemo-56c44849f8-88bm4 2/2 Running 0
14ddemo-56c44849f8-mpngr 2/2 Running 0
14ddemo-56c44849f8-ts6pj 2/2 Running 0 14d

$ kubectl describe pods demo-56c44849f8-88bm4 -n
ui-appName:
demo-56c44849f8-88bm4Namespace:
ui-appPriority:
ONode: ip-10-52-134-48.ec2.internal/10.52.134.48
...
Events:
...
...
```

If the Pod is in a non-healthy state such as Pending or Crashing, then it is also impossible to get into the containers in the Pod for debugging, which is then mainly analyzed by the Pod-level logging information.

Finally, we can go into the container to troubleshoot, and here we will mainly carry out log checks of the service function logic, as follows:

```
$ kubectl exec demo-56c44849f8-88bm4 -c demo -n ui-app --
lsbinenv-configlog

$ kubectl logs demo-56c44849f8-88bm4 -c demo -n
ui-app2021/04/22
03:55:50 proto: duplicate proto type registered: proto.
Int64Slice2021/04/22 03: 55:50 proto: duplicate proto type registered: proto.StringSlice2021/04/22 03:55:50 proto:
duplicate proto type registered: proto.
Message2021/04
/22 03:55:50 proto: duplicate proto type registered: proto.CompanyContacts
...
...
```

If “Can’t connect to MySQL server” appears in the log file, it means that there is a database connection issue and you need to check whether the MySQL configuration of the new online service is correct and whether the network connectivity between the new service and MySQL is normal.

2. Failure analysis for service version upgrade

In a pre-release or production environment, regular version upgrades are successful in most cases. This is because the services have been tested to a certain extent to ensure quality. Most of these problems are caused by the state of the cluster itself and not by the service.

The steps to analyze a version upgrade failure are essentially the same as those described in the first deployment of the service online.

- (a) If we find log: “Warn FailedMount 20s (>496 over 16 h) kubelet, [demo.server.net](#) (combined from similar events): MountVolume.SetUp failed for volume “demo-sftponly”: mount failed: exit status 32 mount.nfs: requested NFS version or transport protocol is not supported,” that means we need to check whether the cluster is using NFS properly.
- (b) If we find log: “[‘Error from server (Forbidden): pods “consumer” is forbidden: User “system:serviceaccount:authorize:demokey” cannot get resource “pods” in API group “in the namespace” “ui-independent-app”’],” that means cluster permissions need to be handled.
- (c) If we find log: “0/4 nodes are available: 1 node(s) had taints that the pod didn’t tolerate, 3 Insufficient cpu.” that means the cluster is under-resourced and needs to be expanded.

The above issue will cause this service deployment to fail and cause the Helm deployment to be recorded as Failed. The current version of Helm used by our team is 2.13.1, which requires the service recorded as Failed to be rolled back to a successful version first and then a new version deployed on a normal record. The Helm rollback command and the prompt for a successful rollback are shown in Fig. 9.27.

3. Failure analysis of the service offline process

Service offline can be used both for the purpose of saving cluster resources and to ensure that the full lifecycle of the application is maintained. The first thing to note is that it must not have a negative impact on the services being provided in the cluster. The main offline process is to first reduce the number of Pods corresponding to the service to 0. If the other services being provided in the

```
helm rollback demo 9 --tiller-namespace ui-app
Rollback was a success! Happy Helm-ing!

helm history demo --tiller-namespace ui-app
REVISION UPDATED STATUS   CHART                                         DESCRIPTION
1      Tue Sep 10 17:49:11 2019 SUPERSEDED demo-6.31.748-rev20190830-1bb852a32fa-aws Install complete
2      Tue Nov 19 11:45:56 2019 SUPERSEDED demo-6.32.944-rev20191112-0e06ebba847-aws Upgrade complete
... ...

9      Tue Mar 17 12:58:59 2020 SUPERSEDED demo-6.34.1295-rev20200304-31a674d065e-aws Upgrade complete
10     Tue Apr 14 11:43:26 2020 FAILED    demo-6.35.1386-rev20200327-1917720e225-aws Upgrade "demo" failed: timed out waiting for the condition
11     Tue Apr 14 13:55:45 2020 DEPLOYED demo-6.34.1295-rev20200304-31a674d065e-aws Rollback to 9
```

Fig. 9.27 Helm rollback

cluster are running as expected over a period of time, the service can be taken offline and dependencies removed using the following command.

```
$ kubectl scale deployment demo-test --replicas=0
```

```
$ helm delete --purge
demo-testrelease "demo-test" deleted
```

In addition, besides the service deployment package and associated dependencies, the Jenkins deployment tasks and corresponding code used need to be removed as well.

9.4 Summary

Based on our team's practical experience, this chapter introduces the concept of continuous integration and continuous deployment and related tools, while discussing the key processes for implementing continuous integration and continuous deployment for the daily development work of software engineers, describing the various configuration methods and inter-pipeline collaboration for continuous integration automation triggers, and illustrating the planning of continuous deployment and implementation of deployment frameworks for multiple clusters in different environments. From a full-service lifecycle perspective, this chapter summarizes the service resource quotas and horizontal scaling in Kubernetes, as well as the service release process and maintenance operations.

This chapter is a practical implementation of DevOps thinking, a solution for development and operations teams. Readers will need to customize their own best practice solutions by continuously adapting and optimizing them in relation to their actual software development process and release requirements.