gRPC Microservices in GO

Hüseyin Babal

MANNING

1 Introduction to Go gRPC microservices

This chapter covers

- Introducing Go gRPC microservices
- Comparing gRPC with REST
- Understanding when to use gRPC
- Applying gRPC microservices to production-grade use cases

Good architecture design and proper technology selection help ensure a high-quality product by eliminating repetitive work and providing the best tool kit for software development and maintenance. While microservice architecture can be implemented in any language, Go is particularly suited for building high-performance cloud-native distributed applications such as microservices in Kubernetes on a large scale. Microservices with gRPC communication have already enabled many companies to implement their products with small services based on their business capabilities and have let those services communicate smoothly with each other and the public. With the help of Go, the distribution of those services becomes easier due to its fast compilation, ability to generate executable binaries,

and many other reasons, which we will see in detail with real-life examples in the upcoming chapters.

gRPC is an open source remote procedure call framework, initially developed by Google in 2015, that helps you to connect services with built-in support for load balancing, tracing, fault tolerance, and security. The main advantage of this framework comes from being able to generate server and client *stubs* (i.e., objects on the client side that implement the same methods as the service) for multiple languages that can be used both in consumer projects to call remote service methods and in server projects to define business logic behind those service methods.

Microservice architecture is a form of service-oriented architecture that defines applications as loosely coupled, fine-grained services that can be implemented, deployed, and scaled independently. The main goal of this book is to provide production-grade best practices for gRPC microservices so that, by the end of this book, you will have the self-confidence to implement the entire system on your own.

1.1 Benefits of gRPC microservices

Within a typical monolithic application, calling different business activities, such as calling a payment service from a checkout service, means accessing a class method in a separate module, which is very easy. If you use microservices, such calls will be converted to network communication. These can be TCP, HTTP, or event queue calls to exchange data between services. Handling network calls is more challenging than calling another class method, which can be managed with a simple error-handling mechanism such as `try-catch` blocks. Even monoliths are easy to use at first, but you may need to decompose them for several reasons, including slow deployments and inefficient resource utilization that affect feature development and product maintenance. This does not mean monoliths are bad and microservices are good; microservices also bring challenges, which we will look at in detail in chapter 2. With the help of gRPC, most of the challenges in microservices, such as handling network failures and applying TLS (Transport Layer Security) to service communications (see chapter 6), can be eliminated. By using these built-in features in gRPC, you can improve both the reliability of the product and the productivity of an entire team.

1.1.1 Performance

gRPC provides better performance and security than other protocols, such as REST with JSON or XML communication, as it uses protocol buffers, and HTTP/2 over TLS is straightforward. Protocol buffers, also known as Protobuf, are language- and platform-neutral mechanisms for serializing structural data, which you will see in detail in chapter 3. This mechanism empowers gRPC to quickly serialize messages into small and compact messages on both the server and client sides. In the same way, HTTP/2 enables the performance with server-side push, multiplexing, and header compression, which we will see in more detail in chapter 5.

1.1.2 Code generation and interoperability

Let's say you have a checkout service and a payment service that allow a customer to check out a basket that then triggers a payment service call to pay for the products in the basket. To access the payment service, you need to have request and response models in some place, such as a shared library, to access them easily. Reusing a shared request and response model seems convenient in microservices but is not a good practice, especially if you are using different languages for each microservice. Duplicating models in a checkout service, typically by creating another data class to build request objects and deserialize response objects into, is a better choice. This is all about preventing an incorrect abstraction, as you may have already heard the statement, "A little duplication is far cheaper than wrong abstraction" (https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction). There is an easier way: choose gRPC to define your messages and generate client stubs so that you can inject this dependency and use it directly in whatever language you prefer. We will dive deep into code generation in chapter 3.

gRPC tools and libraries are compatible with multiple platforms and languages, including Go, Java, Python, Ruby, Javascript, C#, and more. The Protobuf binary wire format, as it travels on a wire like in a network, and well-designed code generation for almost all platforms enable developers to build performance-critical applications while retaining cross-platform support. We will see the details of why Protobuf performs well in interservice communication in chapter 3.

gRPC is getting more popular (https://star-history.com/#grpc/grpc&Date) because you can quickly generate client stubs to provide an SDK of your services within different languages. You only need to decide what kind of business objects you need to have. Once you choose which fields you need for a checkout model, you can introduce respective request and response messages. Remember that those messages are just definitions in IDL (Interface Definition Language) and are independent of any language specification. After you define your message specifications, you can generate language-specific implementations so that any consumer can depend on that source. This also means that the development language on the server side can be different than the client side since server-side methods can be generated as stubs on the client side for specific languages supported by gRPC.

In addition to business objects, you can similarly define service methods and generate implementations. Those service functions can be called after you initialize the gRPC client on the consumer side; again, this client is generated out of the box.

1.1.3 Fault tolerance

Fault tolerance is a system's ability to continue operating despite system failures. An idempotent operation has no additional effect, even if called more than once. Idempotency is key to a successful fault-tolerant environment since you need to be sure that, once you retry an operation with the same parameters in case of failure or not having an expected state, it doesn't change the content of the actual resource. For

example, we may want to retry a user delete operation in case of a network failure on response. If the operation returns the same result even if you call it more than once, we say this operation is *idempotent*.

If an operation is not a good fit for an idempotency use case, you must provide proper validation errors in a response message that helps you know when to stop the retry operation. Once you guarantee this idempotency or proper validation, it is just a definition of the retry policy on the gRPC side. Fault tolerance also focuses on topics such as rate limiting, circuit breakers, and fault injection, which we will see in greater detail in chapter 6.

1.1.4 Security

In most systems, you may need a security layer to protect your product against unverified sources. gRPC encourages HTTP/2 over SSL/TLS to authenticate and encrypt data exchanged between the client and server. More specifically, you can easily set that authentication system up using SSL/TLS, ALTS (Application Layer Transport Security), or a token-based authentication system, which we will cover in more detail in chapter 6.

1.1.5 Streaming

Sometimes you may need to divide response data into several chunks in a paginated way that reduces bandwidth and returns them to the user quickly. Moreover, if users are only interested in specific pages, it is not meaningful to return all the data simultaneously. In gRPC, in addition to pagination, you can also stream this data to the consumer instead of forcing the user to do pagination to get the data iteratively. Streaming doesn't necessarily have to be on the server side; it can also be on the client side or both sides simultaneously, called *bidirectional streaming*. In a typical streaming use case, you open the connection once, and the data is streamed through this opened connection. You will see different kinds of streaming use cases in this book, particularly in chapter 5, when we implement a complete application.

1.2 REST vs. RPC

REST (Representational State Transfer) is a widely adopted protocol for microservices. Still, you may start to think about using gRPC if you have strict requirements such as low latency, multilanguage system support, and so forth. REST is based on HTTP 1.0 protocol that lets you exchange messages in a JSON or XML format between the client and server. On the other hand, gRPC is based on RPC (Remote Procedure Call) architecture that uses protocol buffers' binary format to exchange data over HTTP 2.0 protocol. This does not mean that REST is not compatible with HTTP 2.0; you can set up your REST services based on that protocol with a custom implementation so that it is a built-in feature in gRPC.

Since gRPC has built-in HTTP 2.0 support, you can also use unary and bidirectional streaming between clients and servers, resulting in high-speed communication. With REST services' default settings, multiple client–server communications can introduce a delay in overall system performance.

There are also cases in which REST is more beneficial than gRPC. For example, the REST protocol is supported in all kinds of browsers. Since gRPC support is minimal, you may need to use a proxy layer, such as gRPC Web (https://github.com/grpc/grpc-web), to easily communicate with the gRPC server.

gRPC has lots of advantages, such as being able to define messages to easily exchange data between services. Regarding readability, JSON and XML usage in REST has advantages, such as changing the request freely if there is no explicit business validation for the changed fields. In contrast, you need to follow some rules in gRPC to make a change. We will explain this in detail in chapter 5.

gRPC has a built-in client and server stub generation mechanism for which you need to use a framework in REST such as Swagger Codegen to generate client-side models. This becomes critical, especially once you have multiple services and maintain multiple SDKs for customers simultaneously. Now that we understand the differences between REST and gRPC, let's look at when it makes sense to use gRPC.

1.3 When to use gRPC

If you have strict requirements for browser support, then you need to think of using REST, because you will end up setting up another layer for conversion between HTTP/2 and HTTP/1. However, you can still use gRPC for interservice communication and attach a gRPC load balancer (http://mng.bz/BmZ8) to that service pool to expose API to the public for REST compatibility, which we will see in detail in chapter 9. Other alternatives include Twirp (https://github.com/twitchtv/twirp), an RPC framework built on Protobuf. Twirp lets you enable the REST layer for gRPC services in a way that allows you to access your endpoints, as in the following example, which sends a `POST` request with a JSON payload:

```
curl -X "POST" \
      - H "Content-Type: application/json" \
      -d '{"name": "dev-cluster"}' \
      ➥ http://localhost:8080/twirp/github.com/huseyinbabal/microservices-
      ➥ proto/cluster/Create
```

Polyglot development environments are ideal for gRPC integrations since using the Python client within the Checkout service to access the Payment service, which is written using Java, is very easy with client stub generation. You can apply the same strategy to your SDK generations for public consumers. Also, whenever you change your service definitions, the test fails on the client side, which is a suitable verification mechanism for your microservices.

You will learn how to test gRPC microservices in chapter 7. gRPC may not be the proper selection for simple applications such as startup projects that contain only one to two services since maintaining the proto files that contain service definitions is not easy, especially for inexperienced users.

It is, however, acceptable to use gRPC communication between internal services, but exposing a gRPC interface to customers may not be ideal, especially if there is no SDK for the client for gRPC service communication. If you prefer to expose gRPC without maintaining the SDKs for your consumers, then it is better to share your service definitions with them or provide a clear explanation about how to make gRPC calls to your gRPC services.

1.3.1 Who is this book for?

This book contains many explanations, code examples, and tips and tricks supported by real-life examples that can be useful for the following roles:

- *Developers who don't know Go or microservices*—They can take advantage of starting with introductory chapters about Go, microservices, and gRPC and learn production-grade techniques for gRPC Go microservices. Readers who already know microservice architecture can refresh their knowledge with the resources described in Go, which can be easily adapted to any other language.
- *Engineering managers*—They can improve team developer productivity by adding the best practices described in their playbooks. Applying techniques will introduce good visibility over the entire product that will help to quickly onboard new employees to the team.
- *Software architects*—There are many handy examples and architectural designs that can be potential references for their decisions for new products or features.

1.4 Production-grade use cases

As shown in figure 1.1, we will try to create an e-commerce product in this book with Go gRPC microservices that are automated within a proper CI/CD pipeline and live in a Kubernetes environment. In the following subsections, we'll visit critical parts of the diagram to see how important they are for a typical development life cycle, how gRPC makes those parts easier to handle, and which technologies to use and where.

There will be production-grade examples in this book in the following format:

- A completed project at the end of this book
- Code examples to better understand a specific topic and how it works
- Automation examples, especially with GitHub Actions, to reduce repetitive operations
- Preparing artifacts for deployment
- Security best practices

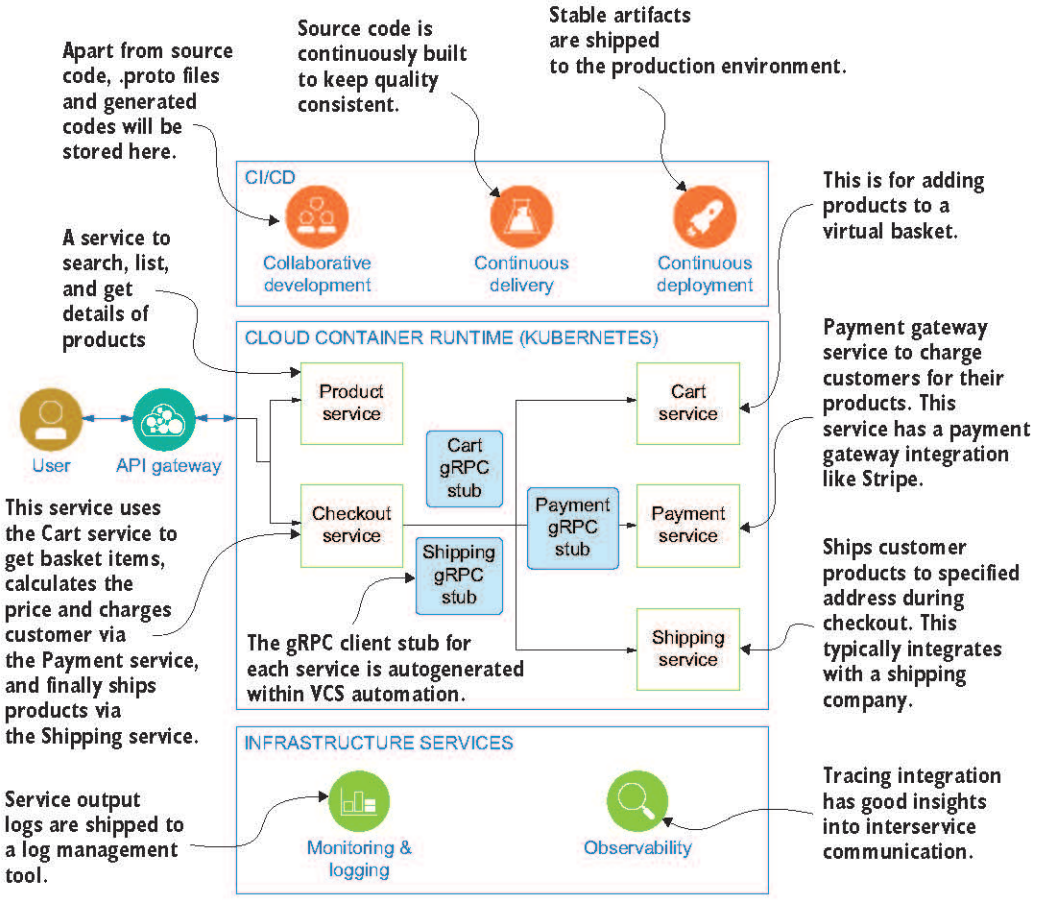

Figure 1.1 Architecture diagram of an e-commerce product built with Go microservices on top of Kubernetes, including CI/CD flow and observability

1.4.1 Microservices

Microservice projects are full of challenges, especially at the beginning of the project, and you will often hear the following questions in your architectural decision meetings:

- Let's implement microservices, but how micro should it be?
- Which strategy do we need to base our construct/decompose services on?

Dividing microservices by business capabilities is one of the options (http://mng.bz/rWnD), and we will use that distinction as we focus on real-life use cases and implement them in upcoming chapters. As shown in figure 1.1, we have five services to provide different business features, such as a Shipping service to ship products to the customer and a payment service to charge a customer's credit card using information in the checkout phase, which is composed of cart items. There are five business

capabilities: product, cart, checkout, payment, and shipping. They connect using their generated stubs (e.g., Checkout uses Shipping gRPC stubs to call Shipping service functions).

Monolith-to-microservice decomposition will replace service function calls with network calls, which means you need to implement a fault-tolerant client for interservice communication. gRPC provides basic things like connection pooling and resource access so that service functions can be accessed using their gRPC stubs on the client side after adding autogenerated stubs to the Consumer service as a Go dependency. As seen in figure 1.1, the Checkout service can call the Cart service to get cart items, the Shipping service to get the customer's address, and the Payment service to charge the customer's credit card by adding respectively generated stubs of Shipping, Cart, and Payment services to the Checkout service as a Go dependency. We will look at dependency management in detail in chapter 5; you will learn how to work with dependencies and how to automate them to generate in a CI (continuous integration) pipeline.

Microservice architecture opens a gate to the polyglot development environment, which is very helpful for choosing the proper language for different use cases. It also allows the use of various technologies such as Neo4j for graph-related use cases, MySQL if there is a need for relational table structures, or Mongo for document-based data models. Microservice architecture also helps you to construct different small teams to assign code ownership to a specific pool of services.

1.4.2 Container runtime

Managing an application environment may not be a real concern if you have a monolithic application because you can deploy this application into a set of virtual machines, and a typical load balancer handles traffic. Inadequate resource utilization, scaling problems, and risky deployments encourage people to move to microservice architecture. However, once you make the switch, because each service is independent, you need to start thinking of a distributed environment that needs proper management.

Kubernetes, an open source container orchestration platform, has already proved itself for application deployment management and many other production-grade use cases. The services shown in figure 1.1 will be all cloud-native applications and will have Kubernetes deployment specs defined for use within a CI/CD pipeline. Moreover, each service will run within a container and can be scaled horizontally based on the load.

gRPC needs a server address to dial in to call service functions. Kubernetes's discovery system is a good fit for finding the correct address because the server address is the service name of a microservice defined within a service spec. Suppose you have a proper naming convention for your services. In that case, you also have perfect integration between the consumer and service, with no help from service discovery products to see the actual address of a specific service.

Each service can have different behavior, such as resource requests, scale factors, language runtime, and so on. Again, they are just configurations within Kubernetes deployments that can be appropriately configured for each service. For example, say the Product service needs more capacity or scaling factors than other services since most customers search and view products during the day. You don't need to simultaneously scale all the services in Kubernetes like you do in a monolithic application. This can be handled by adding scaling factors and resource capacity to specific services.

The main output for each service will be a cloud-native application, which means you can deploy this service to any other container runtime, such as AWS Fargate, AWS ECS, even Docker for local development, and so on, with a little modification.

1.4.3 CI/CD pipeline

There are lots of operations that are candidates for automation within the microservices environment. Service artifact building, gRPC stub generation for specific languages, testing, code quality check, and deployment of services are some well-known examples. The more automation you have for this distributed system, the less stress you have during the development life cycle.

You can easily use gRPC tools to generate stubs on your local environment, but wouldn't it be better to generate them whenever some changes are pushed to the remote repository? You can also generate artifacts to deploy them to an experimental or stable environment after merging them into the main branch. Modern version control system (VCS) providers such as GitHub, GitLab, and Bitbucket already have that kind of integration, so there is not much custom implementation needed for this level of automation.

A green check after a CI/CD job execution does not mean everything is fine; there should be a way to check that the correct mechanism was used. Good coverage of unit tests; proper integration tests to check third-party integrations such as MySQL, Kubernetes, or AWS; contract testing for service-to-service communication; static code analysis; and vulnerability checks are a good start to having a reliable codebase in the main branch.

After a successful and reliable codebase, artifacts can be generated and tagged to be deployed to the user acceptance testing (UAT) environment and then the production environment for end users. Some best practices for deployment methodologies include a rolling upgrade, canary deployment, and blue-green deployment. The main goal with deployment is to ship the artifact, a Docker image in our case, to the Kubernetes environment and be prepared to roll back when needed. The decision to roll back the operation is not easy. Still, if you have a proper monitoring system, you can track error rates and user feedback to decide when to roll back or introduce a hotfix to the current version.

1.4.4 Monitoring and observability

Monitoring is a mechanism that allows teams to watch and understand the state of their systems, and *observability* is a mechanism that enables teams to debug their systems.

Observable systems are achieved mainly through metrics, logs, and tracing. Tracing context is critical to see any specific request's life cycle, which we will see in chapter 9 in detail. Let's say that a consumer uses an SDK to access an API via the API gateway. It propagates requests to four to five downstream services to handle all the operations and then returns to the customer. Having a successful response does not mean everything is good; it is not good if there is a latency within this life cycle. After latency detection, request flows can be analyzed by grouping by trace IDs that contain helpful information. Trace IDs in the requests and response headers can be injected quickly with a simple middleware that we will see in detail in chapter 9.

Monitoring is a crucial part of microservice architecture, because once you decompose a monolithic application into a microservice architecture, you must introduce a solution for better visibility. Service-level metrics, overall latency, and service-to-service call hierarchy are some solutions you may want to see in the monitoring dashboard. In addition to system-level metrics, the logs of the services are also necessary since they allow you to track application-level anomalies such as increasing error rates.

Dashboards, panels, and graphs for your system provide a good start for better observability. Still, we should focus on introducing new metrics and creating specific alarms based on these tools to notify you when you are away from your dashboards. As an example, Prometheus (https://prometheus.io), an open source event monitoring and alerting tool, can be used to collect system and application metrics, and there can be new alert configurations based on those metrics, such as "notify once the memory usage percentage > 80 for a specific service." Logs are also good sources of insight because you can calculate error rates in real time. You can even create alert configurations based on log patterns within modern log management tools such as Elastic Stack (Elasticsearch, Logstash, and other Elastic integration products).

A good monitoring setup can provide insights into both service-to-service communication and service-to-third-party integrations. For example, it will be possible to detect performance problems between a service and a database or a service to a third-party API that is out of the organization's control.

1.4.5 Public access

Public access is important for your product and for your business's reputation. For example, if a user can send unlimited requests to your product, this is a sign of bad architecture design for public access because products without a throttling system can cause resource exhaustion on the server side, negatively affecting performance.

API gateways are widely used to prevent these kinds of scenarios by following certain principles, such as quickly setting up a proper authentication/authorization system, introducing rate limiting to restrict users' request capacity, and so forth. If you already use Kubernetes, you can handle this with built-in features such as adding authorization and rate-limiting configuration to NGINX controllers; otherwise, you have other options, such as using API gateway products.

Resource naming is also crucial because it will affect the quality of product documentation. If proper naming is used for endpoints, it is easier to read the API documentation and consume those API endpoints smoothly. Optionally, you can implement SDKs for your product so that consumers can depend on that SDK feature instead of trying to construct requests, send them to API endpoints, and handle responses.

Summary

- gRPC performs well in interservice communications because it uses binary serialization for the data and transfers it through the HTTP/2 protocol.
- gRPC allows you to engage in client streaming, server streaming, and bidirectional streaming, which gives you the ability to send multiple requests or receive multiple responses in parallel.
- Stable client–server interaction in gRPC microservices is easy because of automatic code generation.
- REST is popular primarily because of its broad browser support, but you can still use a gRPC web proxy (e.g., https://github.com/grpc/grpc-web) for REST-to-gRPC conversion.
- Due to its high portability, Go is one of the best languages for cloud-native applications, such as microservices in Kubernetes.
- Using HTTP/2 over SSL/TLS end-to-end encryption connections in gRPC eliminates most of the security concerns for a microservice.

2 gRPC meets microservices

This chapter covers

- Comparing the advantages and disadvantages of microservice architecture to monolithic architecture
- Understanding communication patterns in microservice architecture
- Analyzing service discovery mechanisms
- How Go and gRPC boost reliable interservice communication and development productivity

The fundamental goal of any software development team is to implement a set of features in order to form a product and create direct or indirect business value. This product can be distributed as a package that can be installed on a computer offline or can be internet based and used online. Each programming language has its own packaging methodology; for example, you can use a WAR or JAR file for Java projects or a binary executable for Go projects. We call this *monolithic architecture*: one or more features/modules are packaged as one product that completes related tasks within a distributable object. When scalability problems arise, alternative solutions like microservice architecture are popular, as the application is

decomposed into services based on their business capabilities. This decomposition enables the deployment of each service independently, which we will see in detail in chapter 8. Interservice communication stability is imperative to providing data consistency among services. This chapter will show how important gRPC is for interservice communication.

2.1 Monolithic architecture

In monolithic architecture, the different components of a monolithic application are combined into a single-tiered and unified software application that can contain a user interface, a server, and database modules that are all managed in one place. Monolithic architecture is especially helpful when developing the initial version of a product, as it helps you get familiar with business domains without having to tackle nonfunctional challenges. However, it is suggested that you assess your product periodically to understand whether it is the right time to move to microservice architecture. Now that we know what monolithic architecture looks like, let's look at its pros and cons.

2.1.1 Development

All modern IDEs are designed to support monolithic applications. For example, you can open a multimodule Maven project in IntelliJ IDEA (https://www.jetbrains.com/idea/) or create a modular Go project with GoLand (https://www.jetbrains.com/go/) that you can easily open and navigate within the codebase.

However, problems can arise as your codebase grows. For example, suppose you have many modules within a monolithic application, and you try to open them simultaneously. In that case, it is possible to overload the IDE, which negatively affects productivity; it also may not be necessary to open them all if you don't need some of them.

Additionally, if you do not have proper isolation for your test cases, you may run all the tests any time you make a small change in your codebase. The bigger the codebase, the longer the compile and testing time.

2.1.2 Deployment

Deploying a monolithic application means copying a standalone package or folder hierarchy to the server or a container runtime. However, monoliths may be an obstacle to frequent deployments in continuous deployment because they are hard to deploy and test in a reasonable time interval. You need to deploy the entire application, even if you introduce just a small change to a specific component. For example, say that we introduced a small change in the newsletter component responsible for serving the newsletter, and we want to test and deploy it to production. We would need to run all the tests, even though we haven't changed anything in other components, such as payment, order, and so on. In the same way, we need to build the system to generate one artifact, even though the changes are only in the newsletter component.

However, deploying a monolithic application might have more significant problems, especially if multiple teams share this application. Flaky tests and broken functionalities other teams introduce may interrupt the entire deployment, and you may want to revert it.

2.1.3 Scaling

Monolithic applications can be quickly scaled by putting them behind a load balancer, which enables client requests to be proxied to downstream monolithic applications in physical servers or to container runtimes. However, it may not make sense from a cost perspective because those applications are exact copies of each other, even if you don't need all the components to be scaled at the same priority level. Let's look at a simple example to better understand this utilization problem.

Let's say that you have a monolithic application that needs 16 GB of memory, and the most critical out of 10 modules is customer service. When you scale this monolithic application by 2, you will end up with 32 GB of memory allocation. Let's say that the custom module needs 2 GB of memory to run efficiently. Wouldn't it be better to scale just the customer module that needs an extra 2 GB of memory by 2 instead of 16 GB?

Distributing monolithic application modules to different teams for fast feature implementation is another challenging scaling problem. Once you decide to use monolithic architecture, you commit to the technology stack long term. Layers in monolithic applications are tightly coupled in-process calls developed with the same technology for interoperability. As a developer or architect, it would be harder to try another technology stack (once one became available). Let's look at the driving factors to scaling next.

2.2 Scale cube

Plenty of driving factors can force you to change your architecture, and scalability is one of them, for performance reasons. A *scale cube* is a three-dimensional scalability model of an application. Those dimensions are *X-axis scaling*, *Y-axis scaling*, and *Z-axis scaling*, as shown in figure 2.1.

2.2.1 X-axis scaling

X-axis scaling consists of running multiple copies of the same application behind a load balancer. In Kubernetes, load balancing is handled by Service resources (http://mng.bz/x4Me), which proxy requests to available backend instances that live in Pod resources (https://kubernetes.io/docs/concepts/workloads/pods/), which we will see in chapter 8 in detail. Those instances share the load, so if there are N copies, every instance can handle 1/N of the load. Some of the main drawbacks of this scaling model are that since each copied instance has access to all data, the instances need more memory than required and that there is no advantage to reducing the complexity of growing the codebase.

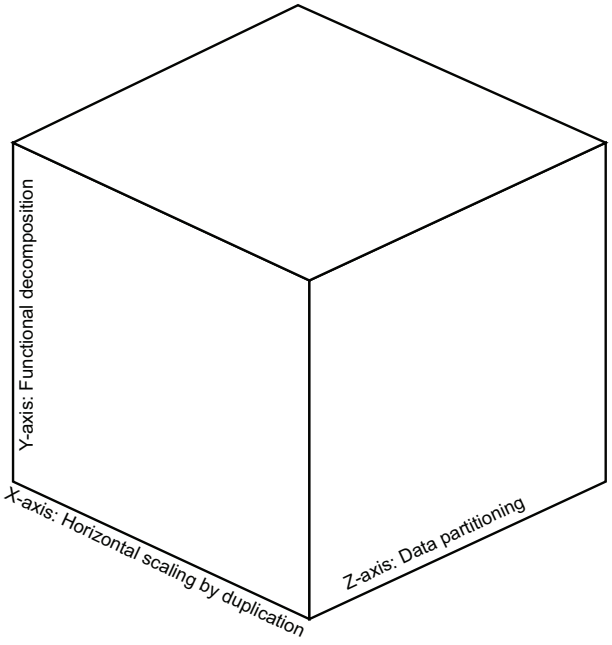

Figure 2.1 Scalability model of an application

2.2.2 Z-axis scaling

Z-axis scaling is like X-axis scaling since the application is copied to instances. The main difference is that each application is responsible for just a subset of data, which results in cost savings. Since data is partitioned across services, it also improves fault tolerance, as only some data will be inaccessible to the user.

Building Z-axis scaling is challenging because it introduces extra complexity to the application. You also need to find an effective way to repartition data for data recovery.

2.2.3 Y-axis scaling

In Y-axis scaling, scaling means splitting the application by feature instead of having multiple copies. For example, you might decompose your application into a set of services with a couple of related functions. Now that we understand the pros and cons of monolithic architecture and scalability models, let's look at how microservices architecture is a form of Y-axis scaling.

2.3 Microservice architecture

Microservice architecture is an architectural style that defines an application as a collection of services. Those applications mainly have the following characteristics:

- They are loosely coupled, which allows you to create highly maintainable and testable services.
- Each of the services can be deployed and scaled independently.

- They are focused on business capabilities.
- Each service or set of services can be easily assigned to a dedicated team for code ownership.
- There is no need for long-term commitment to the technology stack.
- If one of the services fails, other services can continue to be used.

First, you must decide if microservice architecture is well suited for your product architecture. As said previously, starting with monolithic architecture is a best practice because it allows you to understand your business capabilities. Once you start having scalability problems, less productive development, or longer release life cycles, you can reassess your environment to see if functional decomposition is a good fit for your application. Once you decide to use microservice architecture, you might have independently scalable services, small projects that contain specific context during development only, and faster deployments due to faster test verification and small release artifacts.

Let's assume you are familiar with your business model and know how to decompose your application into small services. You will have other challenges not visible in monolithic applications, such as handling data consistency and interservice communication.

2.3.1 Handling data consistency

Having consistent data is essential for almost any kind of application. In monolithic architecture, data consistency is generally ensured by transactions. A *transaction* is a series of actions that should be completed successfully; all operations are automatically rolled back if even one action fails. To have consistent data, the transaction begins first, actual business logic is executed, and then the transaction is committed for a successful case or rolled back in case of failure. As an example, let's assume that once `Order:create()` method is executed, it calls a series of actions, such as `Payment:create()` and `Shipping:start()`. If both `Payment` and `Shipping` operations succeed, it will successfully commit `Order` status as `SUCCESS`. Likewise, if the `Shipping` operation fails, it rolls back the `Payment` operation and marks the `Order` operation as `FAILED` (see figure 2.2).

A typical transaction can be expressed with the steps *begin* and *commit/rollback,* during which you begin a transaction and execute the actual operation; then, you may end up committing data to the data store or rolling back the entire operation. Now that we understand that data consistency can be easily handled in monolithic architecture, let's look at how it is handled in microservice architecture.

2.3.2 Saga pattern

Transactions are a critical part of an application responsible for maintaining data consistency. In a monolithic application, there is a single data source in the same application, but once you switch to microservices architecture, the data state is spread across services. Each of the services has its own data store, which means a single transaction cannot handle the data's consistency. To have data consistency in a distributed system, you have two options: a two-phase commit (2PC) and saga. 2PC coordinates all the processes that form distributed atomic transactions and determines whether they