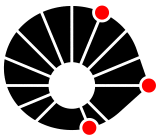

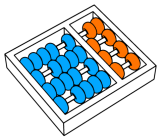

Microservices Design Patterns and Software Evolution

V. A. M. Silva *B. B. N. França*

Relatório Técnico - IC-PFG-20-07
Projeto Final de Graduação
2020 - Agosto

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Microservices Design Patterns and Software Evolution

Aluno: Vitor Alves Mesquita da Silva
Orientador: Breno Bernard Nicolau de França

Abstract - This work explores the impacts of three microservices design patterns on software evolution: API Gateway, Service Discovery, and Externalized Configuration. The main goal was to comprehend the counter effects microservices design patterns have on software evolution. For this, we present an analysis of the impact on software evolution through technical discussions on the counter effects of the design patterns on a couple of evolution scenarios. Results show important aspects of software evolution are impacted by microservices design patterns and emphasize the relevance of comprehending counter effects of microservices design patterns to orchestrate software evolution effectively.

Keywords *- Microservices, Design Patterns, Software Architecture, Software Evolution.*

1. Introduction

The microservices architectural style is rapidly growing in popularity as a way to achieve scalability, availability, and continuous delivery needs considering the fast-paced cycles of modern software development [1]. Microservice architecture is an approach to developing a single application as a suite of small services, each running its process and communicating through lightweight mechanisms [2]. Many technology companies, most notably Netflix, Amazon, and LinkedIn [3], value this approach to software development and choose to deliver their core business through microservices-based solutions. Many others are migrating their existing solutions to a microservice-based one. Despite that, some companies are still hesitant to migrate because they find trouble evaluating the pros and cons [4].

Using architectural and design patterns is particularly important in systems development as they allow developers to reuse known working solutions for recurrent problems [3], creating modular and reusable large-scale applications.

Modularization plays an essential role in a microservice architecture. It allows developers to break down the application into multiple independent and separate services, hence the relevance of studying microservice design patterns.

This work presents some of the most relevant and common microservices design patterns and discusses them under the perspective of software evolution by analyzing the impact of the design patterns in a couple of scenarios. Upsides and downsides introduced by the usage of the design patterns in those scenarios are outlined as a means to analyze the impact on software evolution.

Section 2 presents the background to microservice-based applications and design patterns. Section 3 describes the method for selecting and analyzing the design patterns. Section 4 gives an overview of the design patterns selected for analysis. Section 5 analyzes the impact of those design patterns on software evolution considering a couple of scenarios. Lastly, Section 6 concludes this work.

2. Background

Many microservices design patterns have been defined since the term "microservice" was proposed in 2011, at an architectural workshop in Venice, as a way for participants' to explain the typical architectural styles they've been witnessing in a variety of applications [2]. Moreover, for the complete comprehension of this work, a broader understanding of the concepts of microservices-based applications and design patterns are presented in this section.

2.1. Microservices-based applications

There is no single formal definition of what microservices are and how microservices-based applications should be structured. Still, there is instead a set of principles [2] commonly seen in microservice architectures that, when combined, are sufficient to understand what is a microservice-based application.

2.1.1. Multiple services

The entire application is broken down into a suite of multiple components called services. It allows developers to modify and redeploy each part of the application (service) independently with little or no impact on the rest of the application. This isolation of each service makes it easier to tweak and redeploy the services without compromising the entire application. Still, it comes with the cost of remote calls to other services, as well as a complex operational environment.

2.1.2. Smart endpoints and dumb pipes

One could argue this principle dates back to 1978 [6], the year the Unix philosophy was published, way before microservices were a thing. In the words of Doug McIlroy, the inventor of Unix pipe:

"Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." [7]

An arbitrary microservice should work in a three-step process (Figure 1): receive a request, act upon it accordingly, and produce a response. This way, the whole system is as decoupled and distributed as possible.

Figure 1. Overview of an arbitrary microservice.

2.1.3. *You build it, you run it*

Unlike other development techniques, in which a development team is responsible for a certain segment of the application (e.g. UI, testing, databases, back-end logic), in microservices-based application development, teams tend to be responsible for the entire module they built, which may include a subset of microservices. In the words of the Amazon's Chief Technology Officer, Werner Vogels:

"You build it, you run it." [8]

The underlying motivation originates from the notion that the team's efforts should be on improving the project they have built and, therefore, know better than anyone else.

2.1.4. Decentralization

Due to the heterogeneous nature of microservices architectures, involving a variety of technologies and platforms, a decentralized distribution, governance, and data management is favored against centralized counterparts. Per service instances of databases are preferred instead of centralized database solutions, and decentralized governance gives developers freedom of choice amongst different technologies being able to pick the most adequate tools for the job.

2.1.5. Failure resistance

Dividing the application into multiple components as services enhances the importance of failure tolerance. Most often services communicate through message passing techniques on top of network layers. A network failure may interfere with the communication between services, possibly compromising the functioning of the whole application if the problem is not addressed correctly.

Microservices-based applications put considerable effort into real-time monitoring techniques to recover from failures and provide maximum availability.

2.1.6. Emergent design

An emergent design envisions creating a system that satisfies current needs, but it is also flexible enough to catch up with future developments and technologies. Delivering small changes with business value is at the core of emergent design, focusing on delivering functionalities and adapting the current application rather than following a predefined plan.

2.2. Design Patterns

A design pattern is a general solution to a commonly recurring problem [34]. Many problems in software architecture and design share the same underlying principles and solutions. Design patterns benefit from that commonality and allow reusing developers' knowledge on addressing similar problems in generic well-documented solutions.

Design patterns are often slightly adapted to different contexts and have different names, but the same foundation. A brief overview is enough to understand how important it is to address those commonalities in software design in the form of patterns so the knowledge obtained from one can easily be reused into different contexts. For instance, the Facade Pattern, a software design pattern, shares the same characteristics of the API gateway pattern, a microservice design pattern. The recurring problem is how to encapsulate the application's internal logic and provide an API to its clients. The Facade Pattern solves this problem in the context of object-oriented design, and the API Gateway does so in the context of microservices distributed systems. The similarity can be observed in figures 2 and 3.

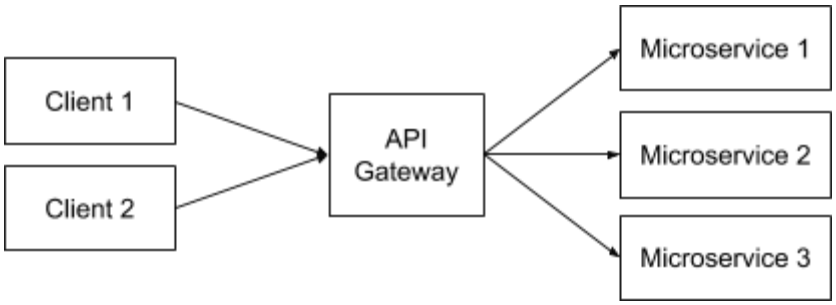

Figure 2. API Gateway pattern example

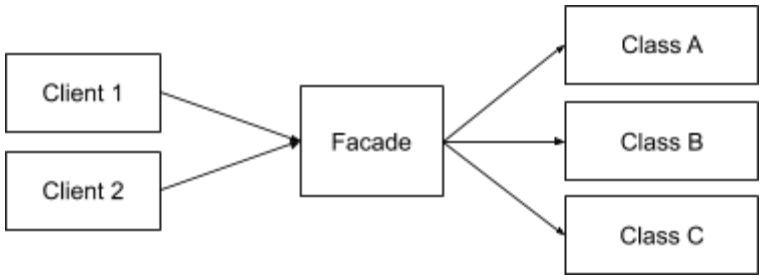

Figure 3. Facade pattern example.

3. Methods

This section presents the methods to perform the study presented in this work.

3.1. Pattern Selection Process

A myriad of microservices patterns has been identified in the literature [9]. Nonetheless, not all of them fit the purpose of this study. Many patterns identified in the literature have little to none use in actual projects, making it difficult to find implementations of the design pattern to be analyzed in existing projects. Besides, some of those patterns' documentations fail to give a clear definition of the problem it is trying to solve and the proposed solution.
In this work, we focus on three criteria for selecting suitable patterns to analyze their impact on the software evolution perspective.

- **C1** - Proven high incidence rate in projects

To analyze software evolution, a high incidence rate in projects is desirable as it allows us to find existing applications and projects to be analyzed. Moreover, C1 ensures that only patterns with relevant presence in actual projects are selected for this study, making it useful for a broader audience.
Márquez and Astudillo [9] conducted a study to determine the incidence rate of design patterns in microservices-based open source projects. Their findings provide a list of sixteen microservices with a high incidence rate in open source projects which was used as the determining factor to choose if a pattern has enough incidence rate to fulfill C1.

- **C2** - The pattern documentation should be clear about the problem it solves.

- **C3** - The pattern documentation should be clear in describing the solution to the problem resolution.

Any microservices design pattern that satisfies those three criteria would be equally suitable for the objective of this study. The sixteen design patterns found by Márquez and Astudillo [9] that satisfy C1 was used as the initial set. All of those sixteen also satisfy C2 and C3 and therefore would suit this study. Among the sixteen patterns, an arbitrary choice of three patterns, with a preference for design patterns from similar domains that tend to coexist and could be analyzed together was chosen to be in the Pattern Catalogue and be studied further: API Gateway, Externalized Configuration, and Service discovery.

3.2. Analyzing the impact on software evolution

The impact of the design patterns in software evolution is analyzed in section 5. The method consisted of identifying the counter effects of the design patterns in some evolution scenarios to determine the impact on software evolution. A description of each evolution scenario was provided and then discussions of the advantages, disadvantages, and challenges the design patterns introduced to the evolution scenarios were outlined, along with an analysis on how to perform the evolution scenario task when the design patterns are in use.

4. Microservice Patterns Description

4.1. API Gateway

4.1.1. Context and problem

Typically, a microservice architecture has several microservices running independently, ranging from one to thousands of microservices. Netflix, for instance, has been reported to have over 700 independent microservices in its architecture in 2017 [10]. But, considering microservices work together to perform various kinds of tasks, a question arises: how do microservices expose their functionality to other microservices and the end-users? A common setup is for each microservice to have its API. However, while directly exposing the APIs for others to use, just as monoliths usually do, is the most straightforward solution. However, it may not be ideal. Take Netflix's case we mentioned before as an example. Imagine having to manage dozens of APIs that have been directly exposed to users and other microservices. Not an easy task for architects and backend developers, who would need to keep consistency across all the APIs, with different teams working on them and for the frontend developers, who would rely on those APIs. In the following, we list some of the downfalls of following that approach.

- **Lack of encapsulation**

External and direct access to services strongly ties clients to application services. In other words, a change in the service's API demands changing client applications to keep things consistent.
While changing an API is an easy task for the backend, the same is not true for the frontend. Mobile users might not update to the latest version immediately and the new version, typically downloaded from app stores, may take a few days to be rolled out to clients because they need to be approved before publishing.
Third-party developers rely on the stability of an API. They might not want or be allowed to update their application to the latest version right away. API stability is something to keep in mind when evolving the microservices application.

- **Invoking multiple APIs might impact application performance**

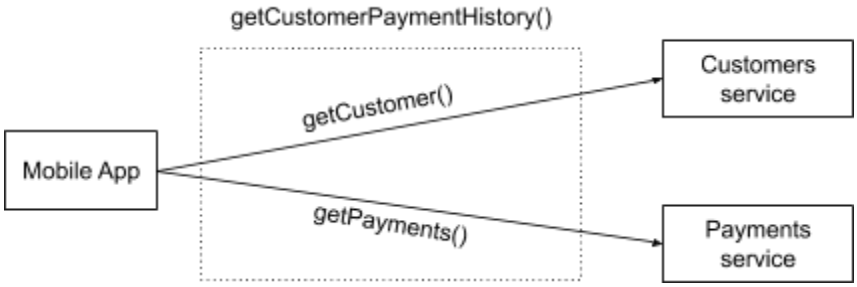

Figure 4. Multiple services invocation example

The first problem of calling services directly is that it leaves the API Composition code to the application developer. However, this should not be developers' responsibility as it could also lead to mistakes as such composition can be complex and APIs may change over time. Figure 4 illustrates the problem showing a *Mobile App* calling *Customer* and *Payments* services, to compose the *getCustomerPaymentHistory* request.
Multiple requests performance can also impact user experience. Whether the API requests are synchronous or asynchronous, the loading time tends to increase in comparison to a single request.

- **Unsupported communication protocols**

Client applications typically run behind firewalls and support protocols like HTTP and WebSockets. Microservices, however, have a broader range of protocols ,and, depending on the protocol, it may not be able to communicate with client applications. Protocols such as gRPC and AMQP may suit better for the microservices internal communication than HTTP

due to performance issues, but client applications would not be able to use them. Either because those protocols are unsupported or firewall policies.

4.1.2. Solution

The API Gateway is an entry point for clients to interact with internal services. As illustrated in Figure 2, the API Gateway stands between the microservices and the outside world, encapsulating internal logic and providing a consistent way for client applications to interact with the microservices.

Requests arrive at the API Gateway that handles them accordingly. Some requests are simply routed to the corresponding microservice or client, but additional functions can be performed as well.

- **Routing**

Routing is the core function of the API Gateway. Received requests are forwarded to their destination according to a routing table.

While it looks simple it's also particularly important as it decouples the clients and back-end microservices.

- **API Composition**

Section 4.1.1 discussed the pitfalls of having client applications directly accessing microservices by making multiple requests. It also mentioned this would imply in the API Composition code being left to the client (Figure 4).

API Composition solves this exact problem. It empowers the API Gateway with the responsibility to receive a single request from the client, call multiple services, merge the results, and respond to the user. This way, client developers no longer have to worry about the composition code, making a single request to the API and retrieve all the required data.

- **Client-specific APIs**

The most straightforward API design choice is to build a one-size-fits-all generic API. However, having heterogeneous client applications consuming from the same API might not be ideal.

Different clients have different needs and behaviors when consuming an API. In Figure 4, for instance, a request such as *getCustomerDetails* that returns all details of the customer might be perfect for a desktop application, running on a high bandwidth low latency network. The same might not apply to a mobile application running on a slower mobile network or an IoT embedded device with limited internet access.

The problem tends to intensify as software evolves and complexity increases. Netflix has reported having issues with a one-size-fits-all API [11], forcing them to take a different

approach and rebuild their API to adapt to the differences of the clients consuming it. In the words of Daniel Jacobson, former director of engineering at Netflix:

> *"Our REST API, while very capable of handling the requests from our devices in a generic way, is optimized for none of them." [11]*

Instead of going for the one-size-fits-all generic API, a better approach is to take advantage of the API gateway pattern capabilities and build client-specific APIs specifically tailored to the client requirements.

- **Protocol translation**

Protocol translation encapsulates internal microservices communication protocols. As discussed in section 4.1.1, protocols such as gRPC and AMQP might be ideal for internal microservices communication but not for the client application communication. Protocols translation in the API Gateway solves this problem by allowing clients to call the API using a widely adopted protocol like HTTP, while the internal microservices communicate using protocols with better performance.

- **Filters**

Filters [13] also referred to as middleware [12] or edge functions [3] are responsible for the business logic of the API Gateway. Filters are request-processing mechanisms and can provide the API Gateway capabilities like authentication, authorization, logging, rate-limiting, DDoS protection, and other request-processing related functionalities.

4.2. Service discovery

4.2.1. Context and problem

Microservices are fundamentally loosely coupled. Usually, the set of services changes dynamically over time. Autoscaling, maintenance, and failures can change the overall structure of the system affecting services availability and network locations.

To communicate with a microservice accessible through an API, clients need the network location, the IP, and port of the service API. This is trivial in a traditional application running in a server as IP and ports are most likely static, however, this poses a challenge in the ever-changing ecosystem of microservices. Oftentimes, the network location of a microservice will change, instances may start or stop, and the other microservices need to be aware of those events to adjust accordingly.

4.2.2. Solution

The main goal of the Service Discovery pattern is to make sure microservices can find each other in the network without the need for static references of IPs and ports. This pattern has to types of mechanisms:

A. Client-side discovery

Client-side Service Discovery methods consist of invoking a service registry to discover the network location of other services (Figure 5). Service Registry's main function is to keep a list of the network location of the application microservice instances.

Services register themselves in the Service Registry so it can keep track of existing services and, when a client needs to communicate with a service, the service discovery asks the Service Registry, gets the location of the service the client needs, and routes the request to the destination service. The client, therefore, does not need to know beforehand the network location of the service to reach it.

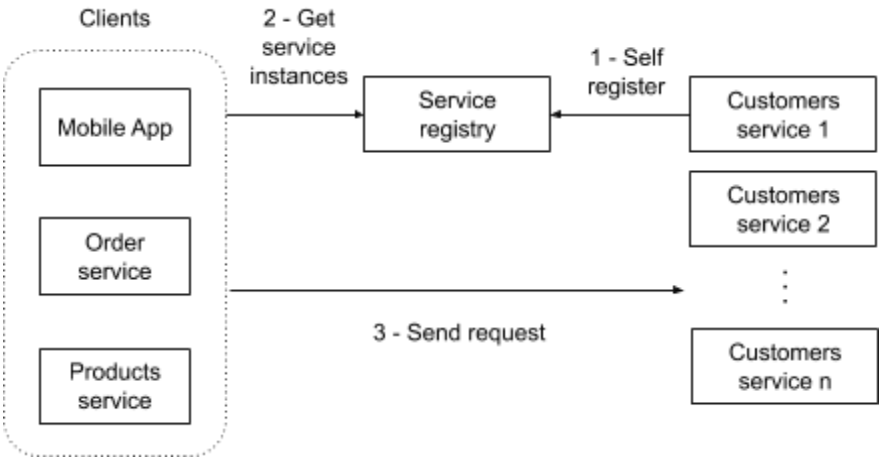

Figure 5. Client-side discovery

B. Server-side discovery

Server-side Service Discovery methods consist of delegating the responsibility of discovering services network location to the server (Figure 6). When a client demands a service, it does not need to know the service network location. A client makes a request to the service alias (DNS name) via a router. The router then invokes the service registry, gets the network location of the destination service and forwards the request.

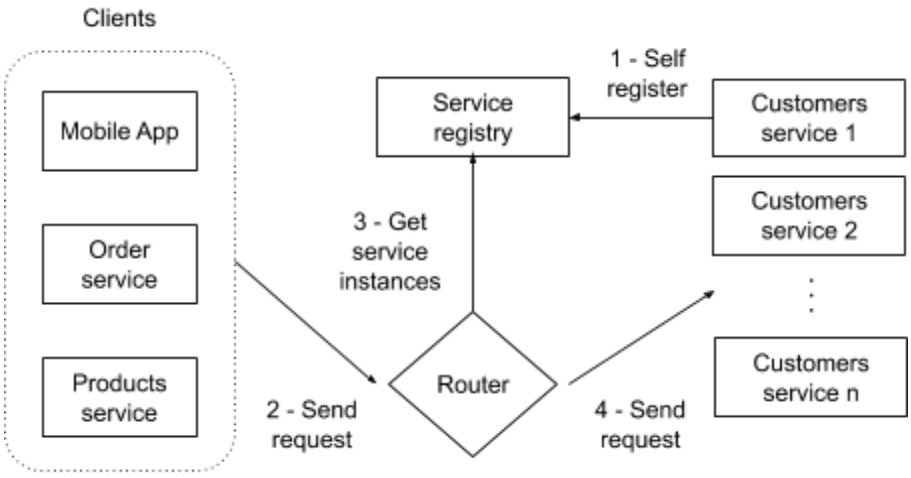

Figure 6. Server-side discovery

4.3. Externalized Configuration

4.3.1. Context and problem

Traditional applications often contain configuration-related information deployed with the application in the form of local configuration files. Changes to the configuration often require the application to be restarted, implying in extra management work and downtime.

Since each application is tied to a local configuration file, it might require additional effort to deploy changes to every configuration file when multiple instances of the application are running, keep track of the changes, and ensure consistency across all instances leading to more management overhead.

Moreover, there are often common configuration settings for every instance of the application, for example, URLs, port numbers, and credentials. It would be simpler if those configurations were shared across the application instances.

4.3.2. Solution

The Externalized Configuration pattern solves the problem by providing an external centralized configuration server for the services configurations.

Services do not require redeploying for configuration changes to take place, ensuring consistency across running instances is much easier as the configuration is centralized at the server, and shared configuration properties can be set up once and be read by multiple services.

There are two main approaches to implement the Externalized Configuration pattern [3]. The push-based model, in which the configuration server (generally built in the deployment

infrastructure) sends the configuration settings to the service, and the pull-based method, in which the service reads the configuration settings from the configuration server.

A. Push-based externalized configuration

The push-based approach consists of sending the configuration properties to the service. The deployment environment (Docker, Kubernetes, OpenShift, etc) is responsible for the configuration server, providing the configuration properties on startup via environment variables or configuration files read by the service instance.

Changing configuration properties of running services, however, is challenging in the push-based method, as environment variables may not be modified during runtime, requiring a restart for the configuration properties update to take place.

B. Pull-based externalized configuration

In the pull-based approach, the service invokes an external configuration server to read configuration properties. The network location of the configuration server will be provided to the service by environment variables (or any other push-based method), and the service will then query the configuration server for the required configuration.

The configuration server can be any external service in which services can get their configuration properties. While specialized configuration servers do exist, for instance, Spring Cloud Config Server, simpler alternatives such as version control systems (Git, SVN, etc), and databases are popular choices for storing configuration properties.

Contrary to the push-based method, changing properties of running services is much easier, as services can implement mechanisms to watch for configuration properties changes in the configuration server and apply the configuration update to themselves accordingly.

The configuration centralization also proves useful in the management of multiple configurations, making it easier to manage multiple configuration properties and ensuring consistency across service instances configuration. In addition, configuration properties can be shared by multiple services, dropping the need for duplicate properties, and reducing complexity.

5. Impact on Software Evolution

This section addresses the impact of the three microservice design patterns (API Gateway, Service Discovery, and Externalized Configuration) on software evolution, considering a couple of scenarios:

- For API Gateway, the migration from monolith to microservices.
- For Service Discovery, the integration with new microservices and serving multiple versions of microservices.

- For Externalized Configuration, configuration updates, and deployment.
- Lastly, for API Gateway combined with Service Discovery, reducing deployment downtime.

5.1. API Gateway

5.1.1. Migration from monolith to microservices

Nowadays, it is common for teams to migrate their existing monolithic solutions to a microservice-based architecture. Existing monolithic applications that might have been thoughtfully crafted a few years back may not suit so well for today's requirements, and the benefits of microservice-based architectures tend to draw the attention of many teams considering a major re-design of their existing monolithic application. Big companies like Groupon [26], SoundCloud, and DigitalOcean [27] all reported that, at some point, they decided to migrate their monolithic solution to microservices.
A popular choice to tackle the challenge of migrating a legacy system to a newer solution is to apply the Strangler Application Pattern, a design pattern that encourages the migration of legacy systems to newer solutions by gradually replacing specific pieces of functionality with the newer solution until the legacy system eventually dies off, or, as the name suggests, is strangled by the newer solution [25].

"One of the natural wonders of this area are the huge strangler figs. They seed in the upper branches of a tree and gradually work their way down the tree until they root in the soil. Over many years they grow into fantastic and beautiful shapes, meanwhile strangling and killing the tree that was their host.
This metaphor struck me as a way of describing a way of doing a rewrite of an important system." - Martin Fowler [25]

An important thing to consider is how to arrange the evolution of the software during the migration process. The API Gateway plays an important role in the migration from monoliths to microservices, as it is the entry point to the system, determining where incoming traffic should be routed to, opening up the possibility to choose between routing traffic to either the monolith or the microservices. Essentially, it works as the so-called "strangler facade" within the Strangler Pattern [35].

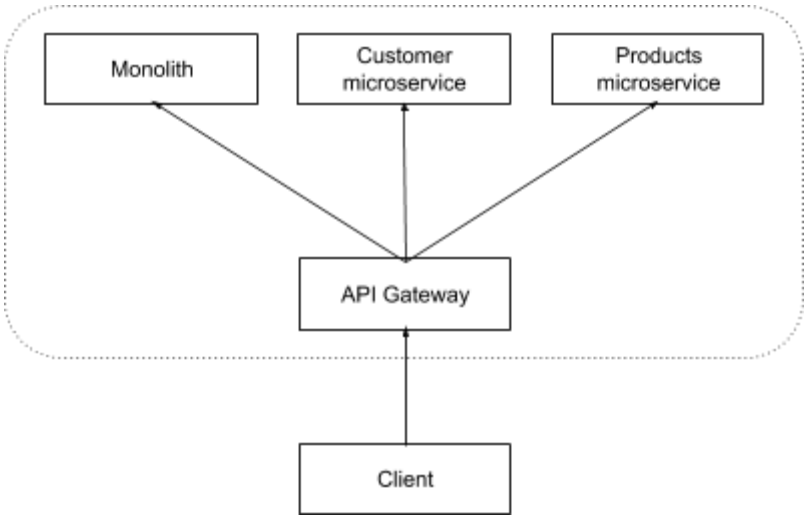

Figure 7. API Gateway traffic routing during the migration process

The API Gateway acts as the proxy layer mapping to the back-end infrastructure. Existing clients are agnostic to changes in the back-end infrastructure, as they only communicate with the API Gateway layer, adding flexibility for back-end changes to be made [35]. And, as software evolves, the monolith can then be broken down, gradually, into smaller pieces packaged into microservices.

Once the choice of using an API Gateway in the migration process has been made, the team should decide where to place it. Should it be integrated into the monolith? Perhaps, it has one already and it could be used. Should it be deployed separately as a service? If so, should it be deployed within the existing infrastructure or within a new infrastructure that will serve as a basis for the future microservices? There is no right answer, but each choice has tradeoffs [36].

Monolith as API Gateway

One option is to add an API Gateway to the monolith or keep using an existing API Gateway from the monolith for the migration process. However, by doing so the monolith becomes highly coupled with the API Gateway.

Redeploying the monolith requires the redeploy of the API Gateway and vice-versa. Adding new routes, for instance, can require a redeploy of the entire application, including parts unrelated to the API Gateway. Also, modifications to the API Gateway should be made to the monolith, but considering the end goal is to strangle the monolith completely, this seems to be some kind of paradox. Why would you keep adding new features to a system you are trying to strangle?

Scalability is also highly coupled with the monolith. The monolith scalability is the API Gateway scalability.
In some cases, though, opting for the API Gateway integrated with the monolith makes sense. Especially, if complex authentication and authorization that depend on the monolith are involved.
Phil Calçado, former Director of Engineering at SoundCloud, has written two publicly available articles on describing SoundCould's migration from monolith to microservices [27],[28], and shared some of the challenges they faced by using the monolith as an API Gateway.

So each request would first hit the monolith, which then would call other services in the background. This strategy worked well for the next few services, but it had several issues. The first problem we detected was that it created some weird coupling between the new services and the monolith, in which changing a service would often require a change in and redeploy of the monolith. On top of that, our monolith was running a very old version of Rails. Without good concurrency we had to rely on serialising requests to all those new services. Our request times were increasing with every new service we added. - Phil Calçado [28]

Separate API Gateway deployed within the existing infrastructure
Another option is to reuse the existing infrastructure, but keep the monolith and the API Gateway as separate systems. The API Gateway can then be developed and deployed independently. At first, this is harder than the previous option (keeping both together), but as the migration to microservices progresses, the API Gateway will become less and less coupled to the monolith. Perhaps, an API Gateway in the existing infrastructure can be reused, making things easier. However, it is important to note that scalability is limited by the existing infrastructure scalability.

Separate API Gateway deployed within a new infrastructure
This can be the most complex alternative, but also the most flexible. Not only this requires the setup of a new service that needs to be set up and maintained, but also the learning curve on how to work with the next deployment infrastructure also needs to be considered, in case developers are not familiar with it.
As the migration progresses, it becomes less and less coupled with the monolith. Working as independently as possible from the monolith and its infrastructure, which is additional complexity, considering the migration end goal is to get rid of them.

Scalability does not depend on the existing infrastructure scalability. It is now limited by the new deployment infrastructure scalability. But, considering microservices deployment infrastructures are built with scalability in mind, that's most likely a good thing.
When it comes to the best location to place the API Gateway there is no right answer, but it is important to be aware of the tradeoffs, as they will impact software evolution differently and need to be chosen carefully.

5.2. Service Discovery

5.2.1. Integration with new microservices

As applications evolve, it is natural that some microservices need to be added and others removed due to new features and improvements. Therefore, the Service Discovery pattern impacts the addition of new microservices. In other words, there is an overhead in the integration of new microservices so they can be added to the service discovery platform.
The analysis of the impact of adding new microservices in a service discovery environment needs to account for the differences between those two methods (Section 4.2.2).
When server-side service discovery clients need to communicate with a service, they simply make a request to a router destined for this service and service discovery business logic is encapsulated into the server. No specialized code needs to be added to the clients, neither to locate services nor to load balance the requests. The server is responsible for it, leaving no overhead for the integration of new server-side service discovery clients [3].
Docker, for instance, uses embedded DNS to provide service discovery for containers. The docker engine has an internal DNS server that is responsible for the name resolution of the containers [21].
Other services can be found by DNS names, demanding no instrumentation code for that. Suppose the “customer” microservice needs to get data in the “products” microservice (namely products-service) REST API. The customer microservice can simply send a request to *products-service*, instead of using the products service IP address, and the docker engine resolves products-service to the VIP (Virtual IP Address) of the products-service.
Load balancing between service instances also needs no specialized client code. Docker in swarm mode is capable of taking care of routing and equally distributing the traffic across service instances.
While the integration of new microservices in server-side discovery is simple and straightforward, requiring no instrumentation, the same cannot be said about client-side discovery. Since the client is responsible for orchestrating the service discovery and load balancing business logic specialized code is necessary, at the risk of coupling the client to the microservices and introducing some level of overhead to the clients [3].

Another drawback is that the service discovery logic needs to be implemented in whatever programming language/framework the client application uses [37]. Netflix Eureka, a JVM based service discovery solution, integrates seamlessly with JVM based clients but may need a sidecar to integrate with non-JVM solutions, such as Spring Cloud Netflix Sidecar, which also needs to be maintained.
The takeaway is that service discovery, especially client-side discovery, does have an impact on the integration with new services, and solutions using it need to account for the overhead this pattern may introduce for this task.

5.2.2. Serving multiple versions of microservices

Microservices under active development evolve. Functionality is modified, bugs fixed, and eventually breaking changes will need to be made. Albeit the developer's efforts to preserve backward compatibility, there may come a time when two or more different versions of the same microservice need to be served at the same time not to allow existing clients to break, especially when third party clients use the microservices, and would be upset if compatibility issues were introduced without their consent. Moreover, multiple versions of microservices can also be observed in an A/B test scenario involving different versions of the microservice. That being said, this section explores how multiple instances of different versions of the same microservice can be served at the same time when using service discovery.
Say a couple of breaking changes are introduced to a “customer microservice” from version v1 to v2, and services and clients that previously were able to communicate with instances of v1 would not be able to communicate with instances of v2. How can the transition from v1 to v2 be done gracefully, so that clients that can only communicate with v1 are not impacted by v2 and vice versa?

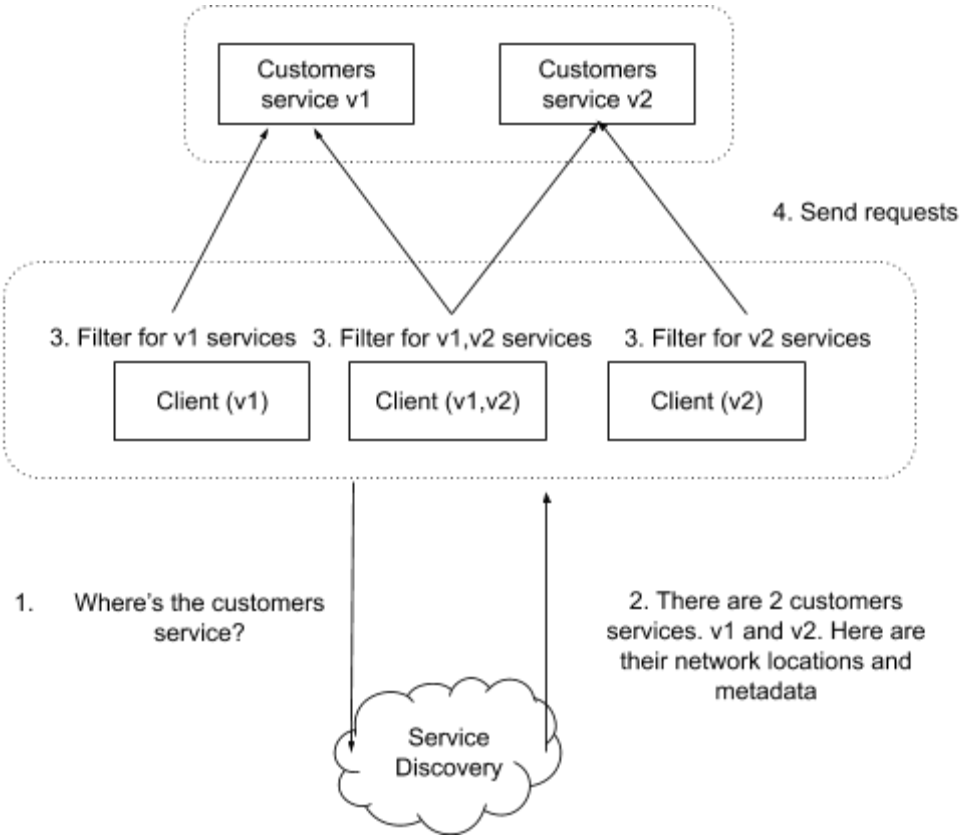

Figure 8. Multi-version service discovery situation

Figure 8 illustrates, step-by-step, how multi-version service discovery works.
Step 1 - the three clients make a request to service discovery asking for the customer service network location and metadata.
Step 2 - service discovery responds with the metadata of all customer services, v1 and v2, such as network location, version, and more.
Step 3 - Clients look at the metadata and filter the customer services by the versions they can communicate with.
Step 4 - Clients send requests only to the microservices they filtered on step 3.

Thit way, multiple instances of different versions of the same microservice can be served at the same time without impacting existing clients, thus improving evolvability. Clients filter by the versions they can communicate to and the rest of the service instances are ignored.
An open-source implementation of this technique is available [17]-[20] and was used as a foundation for this analysis. The author implements service discovery with Netflix Eureka and load balances the requests to service instances with Netflix Ribbon, similar to how was described in this section.

5.3. Externalized Configuration

5.3.1. Configuration updates

Updating configuration properties may pose a challenge as software evolves, since updating properties usually requires an application restart and, in production environments, it is not an acceptable solution.

Runtime configuration updates are nothing new. The concept has been around for years before the Externalized Configuration pattern advent. So, why is this pattern relevant to the topic?

Because it opens the door for easy runtime configuration updates with little dedicated client code to do so, improving the application evolvability.

Spring Cloud Config, a popular externalized configuration software for distributed systems, can be used to achieve runtime configuration updates. For that, service instances can invoke the *refresh* endpoint provided by Spring Boot Actuator to get the latest updates or instead subscribe to a publish-subscribe system at *bus/refresh* with Spring Cloud Bus and the services, listen to events and be notified when they get triggered [39].

Assuming a Configuration Service instance is running, client applications can connect to it and query the *refresh* endpoint for properties annotated with *@ConfigurationProperties* to be updated [38].

```
@Component
@ConfigurationProperties(prefix="customers")
public class PropertyConfiguration {

    private String property;

    public String getProperty() {
        return property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

It is important to note, however, that invoking a Configuration Service API, like in the previous example, is probably enough for most cases, but is not the only solution. Riot Games, for instance, has reported having issues with this approach as their application evolved and the number of configurations grew [14], making REST API calls to the Configuration Service troublesome and opted to migrate to a CDN-type solution with AWS S3. Their configuration service writes the configuration properties to S3 and the services property sources are pointed to S3, instead of the Configuration Service's REST API.
Several implementations of the Externalized Configuration pattern are available, the previous two are only two examples among many more. The takeaway note, though, is that this pattern overall improves the task of updating configuration properties on runtime allowing for greater evolvability.

5.3.2. Deployment

Deployment is part of every application lifecycle and having a well-engineered deployment process is crucial to ensure software evolvability. It is important to realize that the way the software is structured strongly determines how the application is deployed. In the particular case of microservices, in which deployments are frequent, simplified deployment processes are mandatory.
By removing the configuration from the application per se, two main possibilities appear: simplified deployment to multiple environments and service image reusability.

Multiple deployment environments

A common practice adopted in the industry is to have development, staging, and production environments. Would it be wise to manually change configuration properties every time a deploy in a different environment is required? URLs, credentials, and hostnames usually change according to the environment, and manually changing them for every deploy is inconvenient.
With Spring Cloud Config (Section 5.3.1), a microservice deployed into three different environments can have three corresponding configuration files in the server [40] like:

$$Development \rightarrow application-dev.yml$$
$$Staging \rightarrow application-stag.yml$$
$$Production \rightarrow application-prod.yml$$

Client microservices can then connect to the config server during start up and query for the configuration file according to their profile (development, staging, or production). In a

Spring Boot application, for example, one of the methods to set the profile is via the environment variable SPRING_PROFILES_ACTIVE [40], but the concept is extensible to all languages and frameworks and can be seen as an use of the pull-based externalized configuration.

Service image reusability

A popular method to deploy microservices applications is through containers. Container deployment platforms, like Docker and Kubernetes, can create different containers, each one with a different configuration, but all based on the same image.

eShopOnContainers [15] is an example application architectured using microservices. We use it to explore some concepts discussed here.

The application features four API Gateways, Mobile-Shopping, Mobile-Marketing, Web-Shopping, and Web-Marketing (Figure 9). Having multiple APIs Gateways allows multiple development teams to be autonomous when developing and deploying their microservices. In this case, the marketing team can work independently from the shopping team. If a single API Gateway were used, they could run the risk of coupling all microservices to a single part of the application that is developed by different teams [16]. Moreover, the interesting fact about the multiple API Gateways design is that Externalized Configuration was used to reuse the same image for the four API Gateways. They are all based on the same Generic Ocelot API Gateway image, and what determines the functionality of the API Gateways is the configuration.json file, which is applied to the containers through a push-based externalized configuration method. Alternatively, they could have opted for four separate images with integrated configuration properties, one for each API Gateway. However, the images would have to be rebuilt every time configuration updates were made.

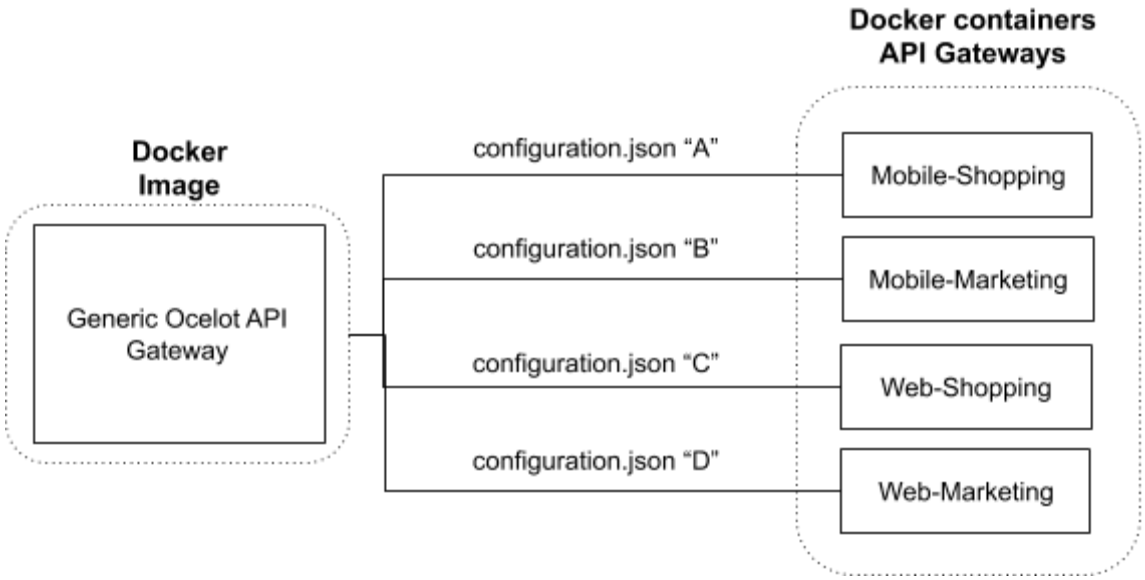

Figure 9. eShopOnContainers API Gateways

The following extract of the docker-compose.yml file shows how the containers are based on the same image.

```
  mobileshoppingapigw:
    image: envoyproxy/envoy:v1.11.1
  mobilemarketingapigw:
    image: envoyproxy/envoy:v1.11.1
  webshoppingapigw:
    image: envoyproxy/envoy:v1.11.1
  webmarketingapigw:
    image: envoyproxy/envoy:v1.11.1
```

The configurations are then pushed to the containers by assigning different docker volumes to the containers that point to the corresponding configuration files as shown next.

```
mobileshoppingapigw:
    environment:
     - ASPNETCORE_ENVIRONMENT=Development
     - IdentityUrl=http://identity-api
    ports:
```

```
      - "5200:80"
    Volumes:
     - ./ApiGateways/Mobile.Bff.Shopping/apigw:/app/configuration

mobilemarketingapigw:
    environment:
     - ASPNETCORE_ENVIRONMENT=Development
     - IdentityUrl=http://identity-api
    ports:
     - "5201:80"
    Volumes:
     - ./ApiGateways/Mobile.Bff.Marketing/apigw:/app/configuratio
n

webshoppingapigw:
    environment:
     - ASPNETCORE_ENVIRONMENT=Development
     - IdentityUrl=http://identity-api
    ports:
     - "5202:80"
    Volumes:
     - ./ApiGateways/Web.Bff.Shopping/apigw:/app/configuration

webmarketingapigw:
    environment:
     - ASPNETCORE_ENVIRONMENT=Development
     - IdentityUrl=http://identity-api
    ports:
     - "5203:80"
    Volumes:
     - ./ApiGateways/Web.Bff.Marketing/apigw:/app/configuration
```

By externalizing the configuration, all functionality of the microservices is determined by the configuration files, reducing the burden of maintaining four separated API Gateway images to a single image.

5.4. API Gateway combined with Service Discovery

5.4.1. Reducing Deployment Downtime

A common setup is to have both API Gateway and Service Discovery working together in highly available services. Service Discovery can be used by the API Gateway to find available service instances, which it can then use to load balance the incoming requests.

Suppose a microservice has two running instances and the API gateway is capable of load balancing the incoming requests between them. *How can a new version of this service be deployed minimizing the downtime*? One way to do it is to shut down the two instances of the microservice, deploy the new versions, and wait for them to startup. But, what would happen with the incoming requests to the API gateway? It would have no available services to forward the requests to and an error would be sent as a response to the client, definitely not something desirable in highly available services. The other option consists of first upgrading one of the microservice instances whilst keeping the other one up, ensuring that incoming requests are fulfilled, and, when the upgrade of the first is finished and the new version is up and has been found by the API Gateway's service discovery mechanism, start the upgrade of second microservice instance by repeating the process.

The second option works best because it allows the application to keep working, but with limited resources. The question that is raised is that of how long the downtime will be, in other words, how long will the application have to work under limited resources. Depending on the application, especially under high loads, having a shortage in one microservice instance can negatively impact some other areas such as the end user experience, so it is desirable that the downtime is as little as possible, ideally being zero.

Unfortunately, zero downtime in the above scenario is not possible, as we will see next, but it is possible to reduce downtime by understanding the delays involved when Service Discovery is being used in conjunction with an API Gateway in such a manner.

The above scenario will be analyzed further with Netflix Zuul as the API Gateway and Netflix Eureka as the Service Discovery platform. The delays can change according to the implementation, but the overall principle stays the same.

Let's first understand the delays involved in the propagation of service instances status to the Service Discovery clients. In this case, Zuul acts both as a load balancer and an Eureka client. Figure 10 illustrates the situation.

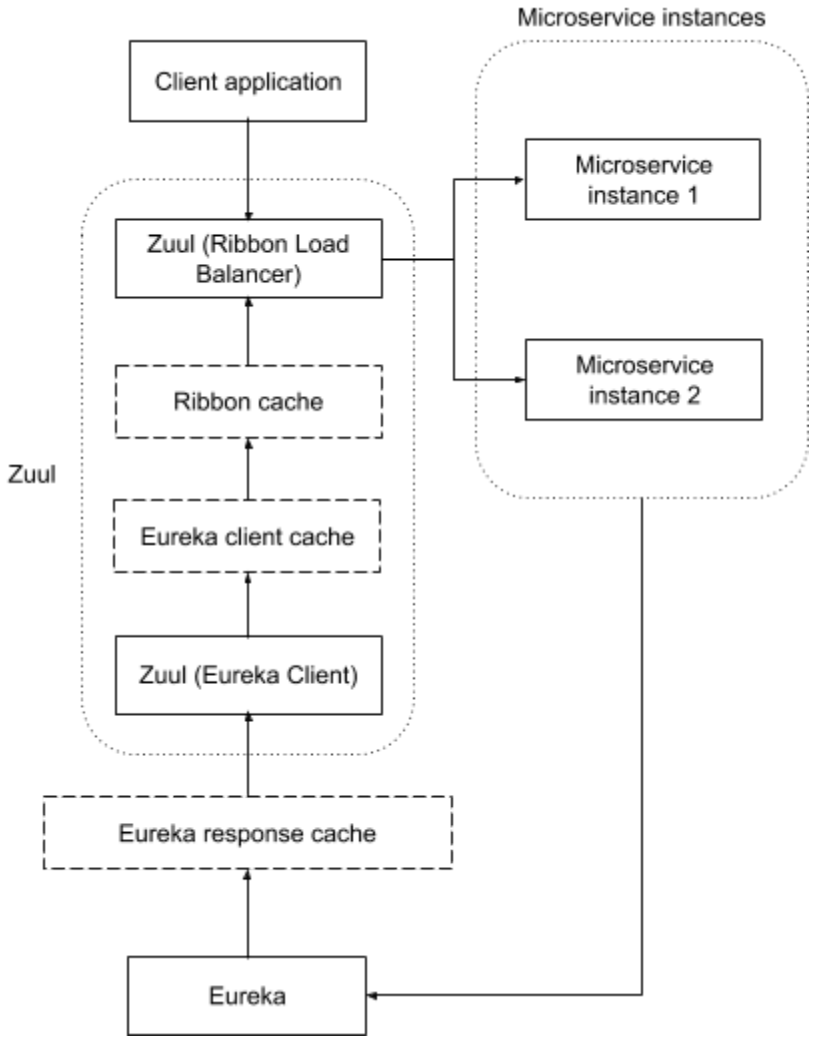

Figure 10. Zuul and Eureka system architecture

The main obstacle to minimize the time of the propagation of microservices status are the caches that stand between the microservice instances and the load balancer. Although they are necessary, they introduce delays in the microservice status propagation process that needs to be taken into thoughtful consideration to ensure both minimum downtime and avoid failures like mistakenly forwarding requests to microservice instances that are not available.

There are many discussions regarding Netflix Eureka cache delays [29]-[33]. The following analysis discusses the same topic, with most of the knowledge coming from said discussions, under the perspective of reducing deployment downtime.

There are three the caches [23],[24] that directly impact the deployment times:

1. Eureka response cache

Default refresh time: 30s

Turns out the response from the Eureka server to incoming requests about existing microservice instances is not based on real-time information. It actually comes from the response cache, which is refreshed once every 30 seconds.

That means that even though the service instance registered with Eureka, it may take up to 30 seconds for this information to be sent to Eureka clients.

2. Eureka client cache

Default refresh time: 30s

To avoid making too many calls to the Eureka server, the Eureka client caches the registry data from the latest request. Meaning it may take up to 30 seconds for the client to update itself with the newest data from the Eureka server.

3. Ribbon cache

Default refresh time: 30s

The load balancer, Ribbon, queries the data it receives back from the Eureka client. Which translates into Ribbon taking up to 30 seconds to get the latest updates from the Eureka Client.

As seen above, the cache refresh times may stack up together because the information is transmitted sequentially through the three caches, each one with a 30 seconds delay. And, coming back to the discussion on the time it takes for the API Gateway to know about status changes in microservice instances, in the worst-case scenario, when the three caches align perfectly, it takes 90 seconds.

The three delays are tweakable and may be decreased with the change of a few configuration properties, but none of these caches can be evicted, so even though the propagation delay can be decreased, the problem persists.

Knowing that avoiding the propagation delay is infeasible, it is necessary to make peace with it and adapt the deployment process to wait 90 seconds to be sure that microservice instances status updates were propagated.

The step-by-step process of upgrading the application is as follows:

1. Shutdown instance 1
2. Deploy the new version of instance 1

3. Shutdown instance 2
4. Deploy the new version of instance 2

What seems to be a simple process is a bit more complicated in the real world due to the propagation delay in the Zuul/Eureka integration.

Shutdown delay

What would happen if Zuul forwarded a request to instance 1 between steps 1 and 2? After step 1 is done and instance 1 has shut down it may take up to 90 seconds for Zuul to be aware of it, so a 90 seconds wait needs to be introduced to avoid forwarding requests to instance 1 while it is down.

Start-up delay

After step 2 is finished, instance 1 is up and running, but it may take up to 90 seconds for Zuul to be aware of it. Not much can be done about it, except for waiting up to 90 seconds for the propagation of the new instance status.

The shutdown and start-up delays into consideration a new step-by-step process to upgrade the application can be defined:

1. Inform Eureka server that instance 1 is going to be out of service soon;
2. In up to 90 seconds, the load balancer stops sending requests instance 1 because it will become out of service soon;
3. Instance 1 waits for 90 seconds to be sure the out of service status propagation has gone all the way to the load balancer and shutdown instance 1;
4. Deploy the new version of instance 1;
5. Wait up to 90 seconds for the new version of instance 1 status be propagated all the way until the load balancer, and start sending requests to the new version of instance 1;
6. Inform Eureka server that instance 2 is going to be out of service soon;
7. In up to 90 seconds, the load balancer stops sending requests instance 2 because it will become out of service soon;
8. Instance 2 waits for 90 seconds to be sure the out of service status propagation has gone all the way to the load balancer and shutdown instance 2;
9. Deploy the new version of instance 2;
10. Wait up to 90 seconds for the new version of instance 2 status to be propagated all the way until the load balancer, and start sending requests to the new version of instance 2.

Summing up the wait times we get a total downtime in the worst-case scenario of 90+90+90+90 = 360 seconds (6 minutes). This downtime needs to be known to choose not

only the best deployment moment (most likely the one with the lowest load) but also to know if this design pattern suits the project requirements.

6. Conclusion

Three microservices design patterns were selected for study, API Gateway, Service Discovery, and Externalized Configuration. Then, each of the microservices design patterns was described by examining the context in which they are used, the problem they solve, and the solution they propose. Next, a couple of evolution scenarios were chosen to analyze the impact of those patterns on software evolution. From the analysis of the impact on evolution, the comprehension of the impact design patterns have on evolution scenarios is crucial to orchestrate software evolution effectively.

Six evolution scenarios were studied, and many insights related to the software evolution that are not obvious or widely known were gathered. However, this is only a fraction of all evolution scenarios that are impacted by microservices design patterns. This, combined with our conclusion on how important it is to understand the impact of design patterns on software evolution is for orchestrating software evolution, indicates the relevance of works on the subject, and is a motivating factor for future works

References

[1] S. Newman, Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2015

[2] J. Lewis and M. Fowler, Microservices-A definition of this new architectural term, 2014. https://web.archive.org/web/20200620232202/https://martinfowler.com/articles/microservices.html

[3] Richardson, Chris. Microservices patterns: with examples in Java. Shelter Island, New York: Manning Publications, 2019. Print.

[4] D. Taibi, V. Lenarduzzi and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," in IEEE Cloud Computing, vol. 4, no. 5, pp. 22-32, September/October 2017.

[5] Fowler, Martin. Patterns of enterprise application architecture. Boston: Addison-Wesley, 2003.

[6] McIlroy, Malcolm D., E. N. Pinson, and B. A. Tague. 1978. "UNIX Time-Sharing System: Foreword." Bell System Technical Journal 57 (6): 1899–1904.

[7] Salus, Peter H. A quarter century of UNIX. Reading, Mass: Addison-Wesley Pub. Co, 1994. Print.

[8] Charlene O'Hanlon. 2006. A Conversation with Werner Vogels. Queue 4, 4 (May 2006), 14–22. DOI:https://web.archive.org/web/20200620232317/https://dl.acm.org/doi/10.1145/1142055.1142065

[9] G. Márquez and H. Astudillo, "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects," 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 2018, pp. 31-40, DOI: 10.1109/APSEC.2018.00017.

[10] Nair, Mayukh. 2017. "How Netflix Works: The (Hugely Simplified) Complex Stuff That Happens Every Time You Hit Play." Refraction: Tech and Everything, blog post, October 17. Available at: https://web.archive.org/web/20200528040543/https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

[11] Jacobson, Daniel. 2012. "Why REST Keeps Me Up At Night." Refraction: ProgrammableWeb, blog post, May 15. Available at: https://web.archive.org/web/20200620231717/https://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15

[12] Traefik, 2020. "Middlewares - Tweaking the Request." Refraction: Traefik, official documentation. Available at: https://web.archive.org/web/20200621041156/https://docs.traefik.io/middlewares/overview/

[13] Netflix Zuul, 2020. "Filters." Refraction: Netflix/zuul, GitHub repository. Available at: https://web.archive.org/web/20200621041627/https://github.com/Netflix/zuul/wiki/Filters

[14] Estes, Leigh. 2016. "The Riot Games API: Deep Dive ." Refraction: Riot Games, blog post, June 23. Available at: https://web.archive.org/web/20200711225828/https://technology.riotgames.com/news/riot-games-api-deep-dive

[15] dotnet-architecture eShopOnContainers, 2020. "eShopOnContainers: .NET Microservices Sample Reference Application." Refraction: dotnet-architecture/eShopOnContainers, GitHub repository. Available at: https://github.com/dotnet-architecture/eShopOnContainers

[16] C. Torre, B. Wagner, M. Rousos. .NET Microservices: Architecture for Containerized .NET Applications. Microsoft Corporation, 2020. https://dotnet.microsoft.com/download/e-book/microservices-architecture/pdf

[17] Otero, Orlando. 2017. "Multi-version Service Discovery using Spring Cloud Netflix Eureka and Ribbon." Refraction: Asimio Tech, blog post, Mar 6. Available at: https://web.archive.org/web/20200716034650/https://tech.asimio.net/2017/03/06/Multi-version-Service-Discovery-using-Spring-Cloud-Netflix-Eureka-and-Ribbon.html

[18] Otero, Orlando, 2018. "ConfigServer." Refraction: Asimio/asimio-cloud, BitBucket repository. Available at: https://bitbucket.org/asimio/configserver/src/master/

[19] Otero, Orlando, 2017. "demo-multiversion-registration-api-1." Refraction: Asimio/tech.asimio.net, BitBucket repository. Available at: https://bitbucket.org/asimio/demo-multiversion-registration-api-1/src/master/

[20] Otero, Orlando, 2017. "demo-multiversion-registration-api-2." Refraction: Asimio/tech.asimio.net, BitBucket repository. Available at: https://bitbucket.org/asimio/demo-multiversion-registration-api-2/src/master/

[21] Rain, Ben. n.d. "Docker Reference Architecture: Universal Control Plane Service Discovery and Load Balancing for Swarm." Refraction: Docker success center, blog post. Available at: https://web.archive.org/web/20200803185455/https://success.docker.com/article/ucp-service-discovery-swarm

[22] Fowler, Martin. 2010. "BlueGreenDeployment." Refraction: martinfowler.com, blog post, March 1. Available at: http://web.archive.org/web/20200615012230/https://martinfowler.com/bliki/BlueGreenDeployment.html

[23] Netflix Zuul, 2020. "Zuul." Refraction: Netflix/zuul, GitHub repository. Available at: https://github.com/Netflix/zuul

[24] Netflix Eureka, 2020. "Eureka." Refraction: Netflix/eureka, GitHub repository. Available at: https://github.com/Netflix/eureka

[25] Martin, Fowler. 2004. "StranglerFigApplication." Refraction: martinfowler.com, blog post, June 29. Available at:

http://web.archive.org/web/20200728040053/https://martinfowler.com/bliki/StranglerFigApplication.html

[26] Geitgey, Adam. 2013. “I-Tier: Dismantling the Monolith.” Refraction: GROUPON Engineering, blog post, October 30. Available at: http://web.archive.org/web/20200728042206/https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/

[27] Calçado, Phil. 2017. “Calçado's Microservices Prerequisites.” Refraction: Phil Calçado, blog post, June 11. Available at: https://web.archive.org/web/20200802234554/https://philcalcado.com/2017/06/11/calcados_microservices_prerequisites.html

[28] Calçado, Phil. 2014. “Building Products at SoundCloud —Part I: Dealing with the Monolith.” Refraction: SoundCloud Backstage Blog, blog post, June 11. Available at: https://web.archive.org/web/20200802234757/https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith/

[29] Renuart, Bertrand. 2015. “Documentation: changing Eureka renewal frequency WILL break the self-preservation feature of the server.” Refraction: spring-cloud/spring-cloud-netflix, GitHub repository, June 9. Available at: https://web.archive.org/web/20200816174306/https://github.com/spring-cloud/spring-cloud-netflix/issues/373

[30] Netflix Eureka, 2018. “Understanding eureka client server communication.” Refraction: Netflix/eureka, GitHub repository. Available at: https://web.archive.org/web/20200816174411/https://github.com/Netflix/eureka/wiki/Understanding-eureka-client-server-communication

[31] COE. 2018. “Eureka_2.” Refraction: MSA Center of Excellence, blog post. Available at: https://web.archive.org/web/20200816174520/https://coe.gitbook.io/guide/service-discovery/eureka_2

[32] Gregston, Taylor. 2019. “Zero-Downtime Rolling Deployments With Netflix’s Eureka and Zuul.” Refraction: Credera, blog post, April 3. Available at: https://web.archive.org/web/20200816174359/https://www.credera.com/insights/zero-downtime-rolling-deployments-netflixs-eureka-zuul/

[33] Mitra, Shamik. 2017. “Microservices Communication: Eureka Client.” Refraction: DZone, blog post, July 17. Available at: https://web.archive.org/web/20200816174401/https://dzone.com/articles/microservices-communication-eureka-client

[34] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design patterns : elements of reusable object-oriented software. Reading, Mass: Addison-Wesley.

[35] Ratnayake, Dakshitha. 2019. “The Truth About API Management in a Microservice Architecture.” Refraction: Medium, blog post, January 31. Available at: https://medium.com/@duckster/the-truth-about-api-management-in-a-microservice-architecture-a3b001a713c5

[36] Bryant, Daniel. 2018. “Using API Gateways to Facilitate Your Transition from Monolith to Microservices.” Refraction: Medium, blog post, Mar 22. Available at: https://blog.getambassador.io/using-api-gateways-to-facilitate-your-transition-from-monolith-to-microservices-5e630da24717?gi=aedf8677cd7e

[37] Richardson, Chris. n.d. “Pattern: Client-side service discovery.” Refraction: Microservices.io, blog post. Available at: https://web.archive.org/web/20200816210614/https://microservices.io/patterns/client-side-discovery.html

[38] Spring Guides. n.d. “Centralized Configuration.” Refraction: Spring.io, blog post. Available at: https://web.archive.org/web/20200816210522/https://spring.io/guides/gs/centralized-configuration/

[39] Spring Cloud. n.d. “Spring Cloud Bus.” Refraction: Spring.io, blog post. Available at: https://web.archive.org/web/20200816210258/https://spring.io/projects/spring-cloud-bus

[40] Spring Boot. n.d. “How-to” Guides.” Refraction: Spring.io, official documentation. Available at: https://web.archive.org/web/20200816213635/https://docs.spring.io/spring-boot/docs/2.2.7.RELEASE/reference/html/howto.html