INSTITUT FÜR INFORMATIK
Betriebssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Refactoring a Monolith into Microservices - An Engineering-Driven Approach

Thorsten Klein

Bachelor Thesis

Start date:	13. October 2017
Submission date:	15. January 2018
Reviewers:	Prof. Dr. Michael Schöttner
	Prof. Dr. Michael Leuschel

Declaration

I, Thorsten Klein, herewith declare, that I wrote this bachelor thesis on my own and did not use any unnamed sources or aids.

Düsseldorf, 15th of January 2018

Thorsten Klein

Abstract

Especially within the past four years the term "microservices" has gained a lot of attention and interest from several software-engineering and -architecture communities. Many discussions were held about whether microservices are replacing the architectural style of monolithic applications which has been the most common way of designing programs for the past decades. In the same discussions, microservices are often talked about as the evolution of Service Oriented Architecture. Also there is much confusion about the actual detailed meaning of the term "microservices" and what aspects and key-characteristics this architectural style bundles or even requires. This bachelor thesis aims to provide a stable base of preliminary knowledge to people that would like to learn about microservices. At the same time it provides guiding information about adapting this architecture, refactoring an existing monolith as well as notes about the usage and deployment of microservices. In order to do so, this thesis will start off giving a brief overview of definitions, history, trade-offs of monoliths, service oriented architecture (SOA) and microservices. This part is primarily focused on exposing the coherences and differences between those architectural styles, while not explicitly targeting SOA. Hand in hand with that goes a brief discussion and some explanations on what a "good" microservice should look like, with some deep insights into the core values of the Twelve-Factor-App. Additionally it will be explained, which reasons urged companies to deprecate their old monolith and why it is still a good idea to start off with a monolithic application instead of directly using microservices in a greenfield development approach. Following the theoretical part that provides information that is useful, before deciding to adapt microservices, the section on methodology will clarify, which organizational and technical requirements are necessary, if the decision has already been made. It will focus on proven approaches to refactor a monolith into microservices as well as on design patterns, operational challenges and useful tools and infrastructure. This includes decomposition-patterns like Domain-Driven Design using bounded contexts followed by establishing a DevOps culture, especially emphasizing continuous integration and continuous delivery/deployment (CI/CD). Likewise, some hints on useful inter-service communication protocols and logging and monitoring will be given. Following the theoretical preparations, a set of previously explained patterns and approaches will be shown using a case-study at trivago, refactoring a legacy monolith written in PHP into smaller microservices written in Go. Doing so, the whole work flow of refactoring the monolith will be explained. Starting with examining the dependencies and module boundaries of the legacy monolith, moving over to preparing the refactoring process, followed by several intermediary design decisions which impacted the final result. Finishing the practical part, the usage of the created microservices and their deployment into the existing trivago-infrastructure will be illustrated focusing on the used tool-set. In the end, the results will be evaluated, also considering the benefits in comparison to the old monolith and a final conclusion on the usefulness of microservices as well as their trade-offs will be drawn.

Contents

1 Introduction

1.1 Motivation

In 2017, software architecture was considered to be the most interesting general IT topic, based on survey results, followed by cloud computing, microservices and DevOps. In a survey about the most interesting software architecture topics, microservices reached the third place in a field that also contained Domain-Driven Design and twelve-factor apps.[Sch17]
Even though, there are several possible ways to architect a software, "the most popular architectures tend to fall into one of two categories: monolithic or microservices".[Cha17] In a fast-paced economy, quick adjustments to new circumstances, new environments and new user-needs require easy and fast refactoring, updating and deploying of applications on a daily, if not hourly, basis. Being bound to an initially chosen programming language and technology stack, it is not easy, often even impossible to switch to newer and faster frameworks, platforms or even languages to improve the application performance. The interwoven nature of modules and the high amount of dependencies that often occur in legacy monoliths make it highly difficult to do any refactoring inside it, without breaking parts of its functionality. With it comes the fact, that it can only be deployed as a single software artifact, making it unusable for quick and easy automated delivery, as each change requires inter-team coordinations before deployment.

With the need for modularization, the Service-Oriented-Architecture (SOA) movement came up to solve some of the above problems by separating logic into services. The problem with this is, that big companies came up with enterprise solutions, introducing the Enterprise Service Bus as the single, centralized communication software used to integrate services. This again centralized a big part of logic into a single part of a system, often even with monoliths instead of small services connected to it only for easier communication.

Working in the direction of decentralizing everything with the single responsibility principle in mind, the microservices movement emerged from a part of the SOA-movement, defining the microservice architecture as a useful subset of SOA. Microservices are nowadays being used to split up existing monoliths into smaller pieces or to create new applications from scratch to avoid the problems of centralized logic. Especially with the growing popularity of cloud platforms, small scale services become a necessity to scale rapidly while keeping the code manageable.

With the successful usage of microservices in large companies like Netflix, Amazon or Spotify, the microservices movement gained a lot of attention and is growing rapidly as public interest increases. Still, adapting microservices requires lots of preliminary knowledge and thoughtful decisions, especially when trying to refactor an existing monolith.

1.2 Objectives

This thesis aims to give a detailed introduction to the definitions of and similarities and differences between the architectural styles of monoliths, SOA and microservices. Highlighting the benefits of microservices with strong respect to the trade-offs coming from the complexity of distributed programming, the first chapters may be used as a quick reference for decision making. In addition to this, different approaches and patterns for adapting the microservice architecture, either in a greenfield or a refactoring approach, are given to provide a coarse overview over the software-engineering possibilities. The practical part of the thesis then serves as an example of applying some of those approaches with a detailed view on decision making before and during the implementation process. To do so, some components will be extracted from an existing monolith and refactored into independent microservices. In the end, the results will be presented, evaluated and compared to the old architecture.

2 Background

2.1 Monolith

Generally speaking, a monolith can be defined as "something having a uniform, massive, redoubtable, or inflexible quality or character" [Dic] or "an organized whole that acts as a single unified powerful or influential force" [Web], while the word itself originates from latin "monolithus" and greek "monólithos" which translates to "made of one stone" [Dic].

In terms of software-engineering, a monolith is almost exactly that: a large and powerful application developed as one unit and built into a single logical executable. For example, in a program that is used as a web-server, all the logic used for handling a basic request runs in a single process. [FL14] Still, a monolith can be component-oriented and have a codebase containing several modules. For example a web application written in Java is packaged as a single WAR file, which might internally consist of several components or modules, but is deployed as one unit. [NS14].

Up to this day, monolithic architecture is the most widely used and most traditional style in software architecture, which allows easy integration of several aspects and capabilities into a single process or application used for several purposes. [Fow14a; Jan16]

Usually, monoliths are written using only one programming language, which the developing team has to choose before starting to design the application. Then, the basic features of that language can be used to split the application up into modules, classes, functions, namespaces or e.g. Java Packages. At the same time, there is generally a tendency to limit the number and diversity of technologies that are used for the program.

When committing a change to the system, the following process involves building and deploying a new version of the whole monolith. The same burden shows up trying to scale a monolithic application according to the current work- or request-load, as it is only possible to horizontally scale it by copying and running several instances of the whole program on different machines. [FL14; Fow14a]

The last two paragraphs already hint on some of the costs that come with a monolithic architecture, which will be discussed in detail later on.

2.2 Service Oriented Architecture (SOA)

Talking about microservices, the question is common, whether this is just SOA, which has already been around and practiced for years. The problem with answering this question is, that on the one hand the term Service Oriented Architecture is not well-defined and can mean many different things and on the other hand that the concept of microservices comes close to what some advocates of SOA dreamed of for this style. [[FL14; RG]]

A service can be described as a well-defined, self-contained and independent function that is not coupled with the states of other services. [Bar17]

As defined by the World Wide Web Consortium (W3C), SOA is "a set of components which can be invoked, and whose interface descriptions can be published and discovered". [SW04] More generally defined, it is a collection of services, that communicate

with each other, where some connecting means is needed. [Bar17]

This means is what causes one of the main differences between SOA and microservice architecture. The main issue that microservice promoters have with SOA is, that usually it is focused on an Enterprise Service Bus (ESB) that is used to integrate several monolithic applications into one environment where they use the Bus for communication. [FL14] Using an ESB aggregates all the logic for routing and transforming data for communication in one central place, whereas microservice enthusiasts prefer using *dumb pipes*. [Thö15] Especially in the late history of SOA, there have been many big projects, including large web-applications, relying on an ESB which could not succeed due to the lack of flexibility using this centralized solution. [Thö15]

Because of these similarities and differences, the borders between SOA and microservice architecture are a bit blurry and most people say that the term "microservices" labels a useful subset of SOA, which is more of a best-practice model than a stand-alone software architecture.[FL14; RG; Thö15; Jan16]

2.3 Microservices

One of the most commonly used and cited definitions of the microservice architecture is the one by James Lewis and Martin Fowler:

> **Quote:**
> "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."
> – James Lewis & Martin Fowler [Fow14b]

This already covers many of the most important aspects and key characteristics in only three sentences. Going a bit more into detail, defining the wording-composition of the term "microservices", the part of "services" boils down to the definition that already came up in the SOA-Section. A service is an autonomous and independent unit, which only executes a single, well-defined task and publishes the output on well-defined interfaces (e.g. via HTTP) that may be consumed by other services. [Jan16]
Against a common impression, the "micro" in microservices does not refer to the actual number of lines of code (LOC) used to implement the functionality. In reality it refers to the scope of functionality, which should be reduced to a single semantic task that represents the smallest possible denominator between implementation and business processes. It also implicates the decomposition of a domain into sub-domains. [RG; Jan16]
Therefore, there is no specified requirement that a microservice codebase needs to meet to be "micro", but the resulting service should only fulfill one single functional, non-functional or even cross-functional requirement that can be understood easily. [Ger16;

Thö15; RG]

In general, there is no fixed and precise definition of the microservice architectural style, but there are some established common characteristics and principles. Those characteristics merge general programming principles with principles that evolved from the Service Oriented Architecture.

Separation of concerns or the **Single Responsibility Pattern (SRP)** are two general key principles in software development, which, in terms of microservices, define the componentization of an application into services that are responsible for a single part of functionality and which are (usually) organized around business capabilities. [HN17; NS14; RG] The commitment to organize a set of microservices around business capabilities or processes is based on a thesis proposed by Melvin Conway:

> **Conway's Law**
> "Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations."
> – Melvin E. Conway, 1967 [Con68]

So the goal is to encapsulate business capabilities as single separated concerns or responsibilities, which makes it easier to reflect changes in business processes in an application. [New15; EI16; Fow14b] An approach on how to achieve this decomposition is given in Domain-Driven Design which will be discussed in a following section.

This decomposition goes hand in hand with two key ideas specific to the microservice architecture that are **Loose Coupling** and **High Cohesion**.

> "Can you make a change to a service and deploy it by itself without changing anything else?"
> – Sam Newman [New15, p. 18]

Loose Coupling emphasizes the independence of services amongst other services. Every microservice should be able to work without depending on the state of any other service and also should not share any data with other services, i.e. it must not share a distributed database or memory (shared-nothing principle.). Consumed data is exchanged between services using well-defined, language-agnostic interfaces (e.g. REST-APIs). As a consequence, the services are independently upgradeable or even replaceable. If a change is made to one service, the update can be built, tested and deployed without having to worry about the functionality of any other service. At the same time this allows one to choose any programming language and technology stack that is suitable for a task, as the APIs should be language-agnostic and e.g. the databases are not shared with others.
High Cohesion is corollary to loose coupling and refers to the bundling of process logic by contexts. Easily explained, this principle declares, that if a change is made in one context, the number of places (services) where changes have to be made for adaption, should be minimal. [New15; Ger16; FL14; Fow14b; RG; Kra16]

Another common characteristic is the contract of **well-defined, explicitly published interfaces** in microservices that can be consumed by other services using a lightweight

communication-protocol like REST[1] over HTTP or STOMP[2] using web-sockets. The protocols used to communicate via those APIs should be language-/platform- and technology agnostic to sustain the flexibility of variable tool-sets for different services. The data might be transferred e.g. in the form of a JSON-Struct or XML. Communication itself may be synchronous, usually using HTTP, or asynchronous, often using AMQP[3]. In case of asynchronous communication, the usually needed message queue is also a good spot for monitoring and logging the communication and workflow of services. [Fow14b; RG; Kra16] As the necessary logic for communication, contrary to the common solution of ESBs in SOA, is moved into the services (endpoints), Martin Fowler and James Lewis named this a concept of "smart endpoints and dumb pipes", where pipes represent the network/queue. [Fow14b]

These are the base characteristics, which the microservice architecture builds on. Further requirements and design patterns for splitting an existing monolith as well as trade-offs of this style follow in the next sections.

2.4 Comparison

With the definitions of those three architectural styles in mind, it is almost obvious that the biggest differences exist between microservices and monoliths. This and the fact, that monoliths are the most used way to structure a program up to this day, make it necessary to compare those two architectures in detail.

Starting with the application development itself, it seems to be clear, that a monolithic architecture is the easiest approach, as it is the most familiar one which has already been used for decades. Also, a monolithic architecture allows to have everything in one codebase what makes inter-module refactoring easier than it would be across several codebases.[Fow14a] But with having everything in one single codebase, several problems may arise.

The first and probably most important fact is, that one **codebase** is usually bound to a **technology stack** that is chosen in the very beginning of development. One language for the project is chosen (may include others, e.g. website would be written using HTML + CSS, etc.) and one storage technology (e.g. MySQL) is defined. As every module of the system is now built using this stack, this commitment makes the project very inflexible, as it becomes almost impossible to ever change a single module to use another stack. Changing to a new stack might be necessary, because the current became too slow or is too old to introduce new features or to find new developers with experience with using it. Even upgrades to a new language level (e.g. Java 8 to Java 9) can cause conflicts because of compatibility problems between modules that use different versions. Due to the decoupled nature of microservices that are developed in separated codebases, there is this degree of freedom that lets one choose the right tool for the job for every service on its own. This feature is also known as **Polyglotism**. [Kra16; EI16; Jan16; FL14]

[1] Representational state transfer: http://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer

[2] Streaming Text Oriented Messaging Protocol: https://stomp.github.io/

[3] Advanced Message Queuing Protocol: https://www.amqp.org/

Example:
With microservices, it is possible to choose e.g. Go with NoSQL for one service and Java with MySQL for another service without having any compatibility problems between those two.

On the other hand, this freedom of choice in technology diversity might as well overwhelm a development organization as it looses track of the used stacks. That's why many big companies, like e.g. Netflix encourage their teams to use a limited (but still big) set of technologies. This also allows for new developers to quickly join the team and adapt to a known technology stack without having to learn a new one every time when moving between teams. [Fow14b; Fow15a; Ger16]

Additionally, a single **large codebase** becomes very inconvenient to use when working together with several teams. Integration of modules developed by different teams and merge conflicts on the shared codebase can be daily problems. When deploying/releasing a monolithic application, the whole codebase needs to be built and deployed. Therefore, pushing a change to one module requires extensive **coordination** between the involved teams and this requires some time, thus slowing down the production velocity. Releases of updates are not that frequent, as teams have to wait for other teams to finish their current update, thus having only one unified release-cycle. [Bou17; Mes16; NS14; Jan16; Kra16]
Microservice architecture in contrast enforces stronger module boundaries, bundling services in separate codebases, which are decoupled from one another. That way, single service changes do not affect other services (as long as they don't change the communicating interface). This loose coupling enables fast and independent updates, upgrades, builds and deployments. Each team owns its full release cycle. Therefore, teams do not have to coordinate changes and releases with other teams, thus decreasing the necessary **communication overhead** which slows down production. [Bou17; Ran16; FL14; Fow15a; Kra16; RG]
Conforming Conway's Law (see section 2.3), microservice architecture represents the communication structure of the development organization by shrinking the coordination overhead and making the development teams more independent. [Fow15a]

This independence of microservices enables the development organization to quickly react to market changes by fast deployment of new versions of required services, without the need to re-deploy the entire application, thus decreasing the **time-to-market**. [Kra16; Fow15a]

As monolithic applications tend to be organized around many inter-connected contexts and therefore often have a deep module hierarchy, new developers usually have some **trouble understanding** the implemented logic and fitting everything into their short-term memory. [FL14; Kra16] Same problems arise during maintenance of an existing monolith, where the developers have to understand the inner workings of the intertwined software architecture. This is usually not a problem with microservices, as a well-implemented service should only fulfill one semantic task and therefore is quite easy to understand. [Jan16]

Microservices also show some benefits in scalability which are also induced by their decoupled nature. **Scalability** is the key for dealing with usage and workload peaks in

applications (e.g. induced by many user requests made to a web-app) and is necessary to prevent overload-induced downtimes.[Jan16]

Excursion: Scaling

There are three types of scaling, that can be represented as the axes of a three-dimensional cube:

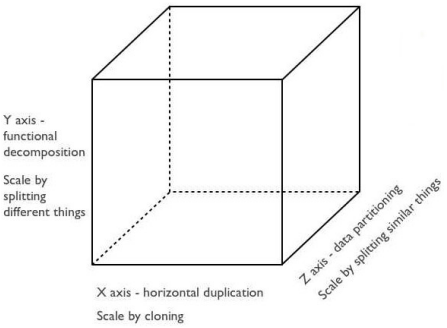

Figure 1: The Scale Cube

As shown in the picture to the left, **X-Axis Scaling** which also known as *horizontal* scaling is achieved by cloning the whole application (onto a new machine) and running multiple instances behind a load balancer. The workload is then distributed evenly over these copies. As a drawback, each copy might access all the data, which is why caching becomes either ineffective or very cost-intensive here.
Y-axis scaling, or *vertical* scaling on the other hand does not create identical copies of the same application but rather runs copies of single services of a decomposed application. Every service is responsible for one semantic task. If the workload for this task increases, vertical scaling allows scaling up only the responsible service instead of cloning the whole application, so it potentially requires less resources.
Using **Z-axis scaling**, similar to horizontal scaling, an identical copy of the application's code is run on different servers. But contrary to X-axis scaling, each server only handles a subset of the data and one component in the system is responsible for routing incoming requests to the appropriate servers. This is usually used for scaling databases. Data partitioning (or sharding) improves cache-utilization, transaction scalability and fault isolation, for the cost of increased application complexity and partitioning scheme. [Ric17c]

As a monolithic system does not provide separate services, vertical scaling is not possible. The problem with horizontal and Z-axis scaling is that they require more technical resources in terms of new and better hardware to improve application performance while not solving the problems of development complexity that arise with growing applications. Microservices by contrast can be scaled independently of one another by applying the vertical scaling approach. This way, only those parts of an application, that are currently under heavy load can be scaled up, while leaving less-used services as they are. This increases application performance while saving resources. Using this approach, one can e.g. scale up CPU-intensive services on the same machine, when cloning the whole application would require to copy it to a new machine, because the memory might be filled up already. [EI16; Mes16; Fow15a; RG; Ger16; Jan16; EI16; Kra16]

Example: Scaling with monoliths and microservices

Thinking of an online-shop, consisting of several services, including e.g. the Shop-Interface showing available items and the Checkout-Service used to actually buy an item. Most of the time, users visit the shop and browse through the list of items, therefore extensively consuming the services of the Shop-Interface, while leaving the Checkout-Service at rest. During usage-peaks, lots of users will browse the Shop-Interface and the applications need to scale to prevent being overwhelmed by the flood of requests. In a monolithic architecture, the whole application including both services has to be cloned onto another server, also copying over the customer-database etc. that is used by the Checkout-Service and others, consuming a big amount of resources. Contrary to that, a microservice architecture allows to only scale the Shop-Interface-Service on its own, maybe even on the same machine without having to duplicate everything.

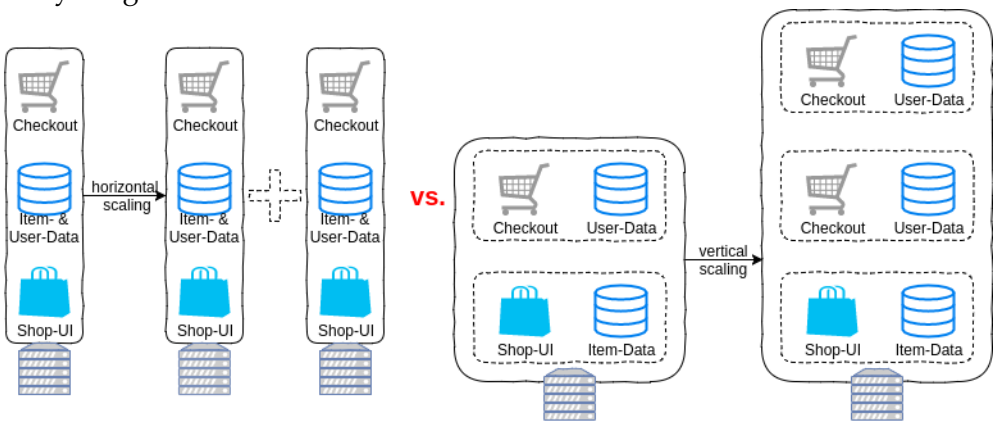

Figure 2: Horizontal Scaling (Monolith) vs. Vertical Scaling (Microservices)

Another benefit of the decoupled nature of microservices is **resiliency**, mainly induced by **fault tolerance and isolation**. While monoliths usually are full of internal dependencies between modules, where every component depends on the state of another component, well-designed microservices work independently. In case of a failure in a single component of a monolithic application, this failure can easily cascade through dependent components and cause an outage of the whole application. For example, a failure in the Checkout-Component could take down the whole Web-Shop, leaving the customer with no interactive product at all. As microservices do not depend on other services' state, a failure in one service does not affect other services or at least does not take them down with it. While a crashing Checkout-Service would cut off the possibility to actually buy something from the Web-Shop, the customer would still be able to browse through the items provided by the Interface-Service. Still, microservices have to be designed in a manner that handles errors (e.g. no responses from a crashed service) gracefully to effectively reduce the blast radius of a failure. This is the downside of the improved resiliency, as it introduces additional complexity to respond to unavailability of a supplier as gracefully as possible. [New15; Fow14b; Kra16; Mes16; Fow15a; RG; FL14]

On the downside, microservice architecture comes with some costs compared to monoliths, mainly in terms of increased complexity.

> "Microservices introduce complexity on their own account. This adds a premium to a project's cost and risk."
> – Martin Fowler, 2015 [Fow15b]

From a development point of view, the microservice approach forces the developers to create a distributed system of services which induce several factors that have to be taken into consideration before starting development. An obvious benefit of monolithic architecture is the fact, that developers only have to focus on a single codebase, where in-process calls (IPCs) are easy to use to communicate between components, with almost no communication overhead or **latency**. Because microservices are decoupled, IPCs are not possible and have to be transformed to Remote Procedure Calls (RPCs), which are sent over the network. The fact, that every service query has to be serialized and sent over the network introduces the risk of high latencies compared to (instant) IPCs in a monolith. The developing team has to take care of graceful handling of errors caused by timeouts and network partitioning and has to implement efficient **inter-service communication** mechanisms. While the performance-costs caused by the latency of remote calls can be mitigated by increasing the granularity of calls and using asynchronous communication, this itself comes at another cognitive cost. And still the unknown **reliability** of remote calls can cause problems that cannot be mitigated, thus the application has to be carefully designed for handling failure. [NS14; Ran16; Fow15a; Kra16; Fow14a]

In addition, when developing a set of microservices, first an extensive knowledge in the business processes, which the system should reflect, is necessary. Only this way, it is possible to find and define the strong **modular boundaries** that define the microservices. At the same time, the knowledge is crucial to create well-defined contracts for developing the APIs, which used for communication between services. This challenge will also be discussed in an upcoming chapter about Domain-Driven Design, because, the better the understanding of a domain is, the easier it is to find proper bounded contexts for the services. [New15; Bou17]

With using a distributed system of services, also data management using a database becomes a problem. Because data needs to be consistent between all the nodes, some mechanism for **distributed transactions** ("notoriously difficult to implement"[FL14]) or for ensuring **eventual consistency** (recommended: "microservice architectures emphasize transactionless coordination"[FL14]) has to be implemented. Otherwise, inconsistencies in the distributed database would cause misbehaviors and failures in the system. This introduces another complexity for software developers as they have to catch errors of the service in case it tries to make decisions on inconsistent data during an inconsistency window. Within a monolith, in contrast, several updates can be managed with a single transaction to one database, without loosing consistency.[Bou17; Fow14b; Fow15a; Fow15b; EI16]

Distributing every aspect of a system also makes testing more difficult, as the interaction between services has to be simulated when testing a single one of them, while latencies, etc., that can affect the functionality, are random. At the same time, a new test has to be written for every service instead of one for all. Then again, this problem can be mitigated by early establishment of Test-Driven Development in a team. [Bou17; EI16]

Lastly, an implementation of a set of microservices requires a lot of **automation** in terms

of testing and deployment. Establishing a culture of continuous integration and delivery/deployment (**CI/CD**) is vital for adapting microservices. DevOps (Development + Operations Engineer) is a keyword that often comes up when talking about automation in a microservice environment. Establishing this culture requires big changes in the organization's culture and development/operations infrastructure, as e.g. a CI/CD-Pipeline has to be introduced. One reason for this is the sheer amount of services that has to be built, tested and deployed on a regular basis by several independent teams, different to a monolithic approach, where only one single application has to be released every other day. This topic will be discussed in detail in section 3.4. [New15; EI16; Bou17; Fow15a; Fow15b; EI16]

Other challenges, when adapting microservice architecture, that are less important in a monolithic approach, include the **orchestration** of many containers, address **discovery** and service **monitoring** as well as **log-aggregation** using GUIDs for the different allocations. [New15; Bou17; EI16] The characteristics and expressions of these aspects are as well featured in section 3.4.

2.4.1 Table-Overview

The following table gives a brief overview of the most important aspects that differ between the monolithic and the microservice architecture:

	Monolith	*Microservices*
	Development	
Velocity - Coordination [Bou17; Ran16; FL14; Fow15a; Kra16; RG]	- **complex** - high coordination efforts - teams blocking each other - teams depend on each other	- **simple** - minimal communication/coordination overhead - independent teams
- Updating Tech Stack [Fow14b; Fow15a; Ger16]	- **complex** - bound to initial decision on technology stack and programming language - single storage technology - thwarting legacy technologies	- **simple** - free choice of tooling/framework/programming language for each service independently - polyglotism - accelerating new technologies
- Release [Fow15c; Thö15]	- **simple** - short initial time-to-market	- **complex** - longer initial time-to-market - more planning required beforehand
Design - Code-Base [Bou17; Mes16; NS14; Jan16; Kra16]	- **complex** - one large codebase - inter-module dependencies - deep directory hierarchy - complex to understand - covers many contexts/tasks - (blurry) module boundaries	- **simple** - several smaller codebases - no inter-service dependencies - flat directory hierarchy - easy to understand - covers a single context/task - well-defined service boundaries/APIs

	Monolith	*Microservices*
- Communication [NS14; Ran16; Fow15a; Kra16; Fow14a]	- **simple** - in-process calls (IPC) - no latency - fail-safe/reliable	- **complex** - Remote Procedure Calls (RPC) via network - higher latency - not fail-safe - needs careful design for failure
- System-Design [NS14; Ran16; Fow15a; Kra16; Fow14a]	- **simple** - one application - centralized	- **complex** - several services - distributed
- Testing [Bou17; EI16]	- **simple** - one application - IPCs only	- **complex** - several services - consider latencies/failures of RPCs
- Data-Storage [Bou17; Fow14b; Fow15a; Fow15b; EI16]	- **simple** - one technology - centralized - ensured consistency via multi-update transactions	- **complex** - several technologies - decentralized (per service) - distributed transactions (discouraged) - eventual consistency (preferred) - avoid inconsistency windows
- Monitoring/ Logging [New15; Bou17; EI16]	- **simple** - Application-Level Monitoring and Logging	- **complex** - Application- + Container-Level Monitoring - Distributed Log-/Metrics-Aggregation
Operational		
Deployment - Process [New15; Ger16; FL14; Fow14b; RG; Kra16]	- **simple** - single application - transfer one codebase - slow coordinated updates	- **complex** - several applications - transfer several codebases - frequent uncoordinated updates
- Coordination [Bou17; Ran16; FL14; Fow15a; Kra16; RG]	- **complex** - team coordination necessary - whole application (one time)	- **simple** - team coordination not necessary - single services independently (any time)
- Updates [Bou17; Mes16; NS14; Jan16; Kra16]	- **neutral** - low frequency - coordination required	- **simple** - high frequency possible - no coordination required - independent
- Addressing [New15; Bou17; EI16]	- **simple** - Discovery/Registry not needed	- **neutral** - Service Discovery/Registry required

	Monolith	*Microservices*
Usage in production - Error Isolation [New15; Fow14b; Kra16; Mes16; Fow15a; RG; FL14]	- **complex** - Single Point of Failure - cascading failure can cause outage - blast radius of component failure = whole application	- **simple** - failure isolation - no cascading failures = less outages - blast radius of service failure = only concerning service
- Scalability [EI16; Mes16; Fow15a; RG; Ger16; Jan16; EI16; Kra16]	- **complex** - inflexible - resource-intensive - X-axis cloning (whole application) - Z-axis sharding (whole application)	- **simple** - flexible -resource-saving - X-axis cloning (whole system or single services) - Z-axis sharding (whole system or single services) - Y-axis vertical scaling (single services)
- CI/CD-Pipeline [New15; EI16; Bou17; Fow15a; Fow15b; EI16]	- **simple** - not required - can be done manually - single application only	- **complex** - required - 100% automation required - has to handle many services

Table 1: Comparison: Monolith vs. Microservices (Overview)

3 Methodology

3.1 Adapting Microservice Architecture

Before using the microservice architecture for a new project, a development organization should take a few aspects into consideration.

Greenfield development is very difficult and especially challenging when considering a microservice architecture, due to the required infrastructure and due to the challenges and necessary preliminary knowledge in business processes, domains, bounded contexts, etc. [New15] Because microservices are quite challenging when developing a new system that comprises several functionalities, experts like James Lewis or Martin Fowler recommend a Monolith-First approach.[Fow15c]

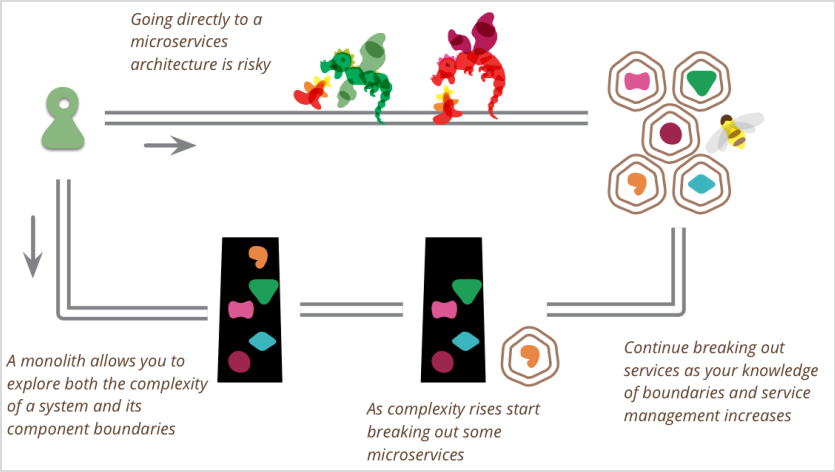

Figure 3: Monolith First

"You shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile."
– Martin Fowler, 2015 [Fow15c]

This approach recommends building a functional monolith first to get to know the domain and processes before splitting it up into separate microservices. Using this approach exposes some benefits. On the one hand it gives time to learn more about the domain and processes and on the other hand it speeds up the initial time-to-market of a product, as it is built in a traditional manner without having to plan a distributed system of services first. Not only is this caused by easier development, but also by uncomplicated deployment of a monolith that can happen in a single step by compiling/transferring the sole codebase. [Thö15]

Knowing beforehand, that a system will grow big and complex enough to significantly outshine the incurred "Microservice Premium" (Fowler), it is a recommended approach to focus on modularization from the very beginning. Separation of concerns, e.g. via using Model-View-Controller (MVC) and the "Don't Repeat Yourself"-Principle (DRY) help keeping module boundaries tight and clear from the start, thus making it easier to

decompose the monolith into microservices later.[Fow15c; RG]

In his "MonolithFirst"-Article, Martin Fowler introduces four different approaches to start off with a monolith. The first approach is about carefully designing a monolithic application with special respect to modularity, making it relatively easy to decompose it into microservices later on. The next approach also starts with a monolith, where the microservices are "peeled off" gradually while leaving "a substantial monolith at the heart" [Fow15c]. As a third approach, Fowler mentions the total replacement of an existing monolith designed as a "sacrificial architecture" that was only used to shorten the time-to-market for a new product. Finally he proposes to start with coarse-grained services which can be refined into finer-grained services once the boundaries stabilize and the team got used to managing several services. [Fow15c]

> "Don't start with a monolith if you want to end up with microservices. It's usually a good idea to follow the YAGNI principle, but a monolith won't magically contain services waiting to be liberated."[a]
> – Stefan Tilkov, 2015 [Til15]

[a] YAGNI: http://wiki.c2.com/?YouArentGonnaNeedIt

Other people like Stefan Tilkov argue against these ideas, that starting with a monolith prevents a team from getting used to microservices from the beginning and can take to much effort and discipline to "build a monolith in a sufficiently modular way" that it can be easily broken down into separate services.[Til15] In the same breath, he argues, that splitting up an existing monolith is very difficult, as the parts in a monolith are, per definition, coupled together very tightly with lots of internal connections and dependencies.

While there are strong arguments on both sides, supporting and resisting against a Monolith-First-Approach, both movements usually align on a common opinion about considering microservices for a new project, that is summed up in the following quote, based on the considerably high costs of a microservice architecture:

> "Don't even consider microservices unless you have a system that's too complex to manage as a monolith."
> – Martin Fowler, 2015 [Fow15b]

3.2 Microservice Design Guideline

When planning to build an application using the microservice architecture, many factors have to be considered before starting with the design blueprint. Those considerations have to include several decisions in the fields of Data Management, Communication (including communicational style and discovery), Security, User Interface, Observability, Service Decomposition as well as Testing and Deployment. [Ric17a]
Searching for helpful tips to design a good microservice, many sources repeat the same principles over and over again which are already established in the microservice movement. Next to the shared-nothing and single-responsibility principle, Netflix, as one of the pioneering microservice practitioners, especially emphasizes the **fail-fast-fail-often**

ethos, which implies, that good microservices should be designed for failure from the very beginning, as it can not be prevented. [Jan16]
Design for failure means, that applications should be designed in a way, that they tolerate and gracefully handle the failure of other services. As any service could potentially fail e.g. due to the unavailability of a supplier, the consuming service needs to handle this as gracefully as possible to repel negative effects to the whole system. This additional complexity requires extensive testing, also in production to ensure the maximum availability and quality of a web-service. [FL14]

> **Example: Netflix' Simian Army and Chaos Monkey**
> Netflix open-sourced a suite of tools called Simian Army that includes a software called Chaos Monkey that is actively used in Netflix' production environment. Chaos Monkey is a resiliency tool that randomly induces instance failures in Netflix' production system of microservices to verify the failure tolerance of their software. In the same run, it also challenges the monitoring and alerting toolset, which is used to check the health of services. [Net17; FL14]

The **shared-nothing principle** refers to **context-separation** between services and also enforces the separation of databases as well as reducing code-duplication by demanding Remote Procedure Calls via APIs. [Jan16]
Single Responsibilities per service are achieved by focusing on **high cohesion**, meaning to aggregate similar or connected functions in one service in such a way that changes required for the same reason, only have to be applied in the same location. [Jan16]
The principle of **loose coupling** is usually mentioned in one sentence with the two principles explained above. While, inter alia, making independent upgrades possible, loose coupling also underlines the need for consistency of communication by using guaranteed consistent APIs. [Jan16]
Before starting to design a set of microservices it is good practice to first have a look on best practices and approaches established by successful practitioners of microservice architecture, like Netflix, Amazon, eBay or others. One of the more general tips published by some of those companies is to think **future-oriented**, especially when designing interfaces and APIs. This advises to use open and platform-independent interfaces that are already established as de-facto-standards, which will be further developed and used in the future. Also, the microservice movement emphasizes the usage of **open-source** software and even the big companies mentioned above strive to contribute to the open-source community. [Jan16]

While the above principles are some good points to always keep in mind while developing microservices, they do not provide a detailed guideline. The Twelve-Factor-App "is a methodology for building software-as-a-service [(SaaS)] apps" [Wig17], that declares a list of twelve factors that need to be considered when building SaaS-Apps, especially Web-Apps. The list of factors was created by developers working on the Heroku platform, "a cloud platform based on a managed container system, with integrated data services and a [..] ecosystem, for deploying and running modern apps" [Sal]. Additionally, the authors of the 12-Factor-App already made extensive experiences with building and deploying lots of apps and their work on the 12-Factor-App was "inspired by Martin Fowler's books

Patterns of Enterprise Application Architecture and *Refactoring*". [Wig17]
Even though, this list of factors was not specifically designed for the development of microservices, the authors' intentions match most of the intentions of the microservice architectural style. Those intentions include optimization for setup automation for minimizing necessary onboarding efforts for new developers joining the project and optimiziation for maximum portability between execution and development environments. Also, optimization for deployment on cloud platforms, continuous delivery and deployment as well as easy scaling of the software are key goals of the 12-Factor-App. [Wig17]

As the Twelve-Factor-App made up a major thematic anchor and continuous reference for the implementation part of this thesis, an enumeration of the twelve factors, including their abbreviated explanations, is given below:

1. Codebase:
   One app is developed in a single codebase that is tracked in revision control (e.g. Git) and if there is more than one codebase it is not an app but a distributed system where each component is an app.
   Also code should not be shared between apps and if it is necessary to do so, the shared code should be factored in a library that can be included using a dependency manager.

2. Dependencies:
   Dependencies should be declared and isolated explicitly, using a dependency management tool. Imported libraries should never rely on system-wide installed packages but rather be bundled in the application (vendoring/bundling).

3. Config:
   Every variable that might change between deploys or execution environments is treated as a configuration that must not be hard-coded or stored in a file within the tracked codebase. The platform- and language-agnostic optimum is to store every piece of configuration in the environment, managed independently for each deploy.

4. Backing services:
   If a service is consumed by an app over the network to feature some functionality, it is called a backing service and should be treated as an attached resource. Those attached resources are accessed using URLs or other locators etc. that are stored in the configuration in the environment. They are loosely coupled in a way that allows them to be exchanged with a different backing service anytime without changing anything in the app's code.

5. Build, release, run:
   Any change to the codebase must create a new release that runs through the build-, release- and run-stage for deployment. The build stage converts the codebase into an executable (build). The release stage combines this with the config attached to the current deploy and thus enables it to be executed by the runtime (run stage) in the execution environment. While builds are developer-initiated, runtime execution can also happen automatically, e.g. after a server reboot.

6. Processes:
   A Twelve-Factor app is executed as one or multiple processes that are stateless and

follow the shared-nothing principle, where any necessarily persistent data is stored in a stateful backing service (database). Sticky sessions, that assume, that another request is served by the same process as the one before are discouraged and the app should not assume to have access to any cached data.

7. Port binding:
   The app should be truly self-contained and not rely on an external webserver that is injected at runtime. Exported services are published via HTTP by binding to a specified port and listening for incoming requests that are routed in by an upstream routing layer.

8. Concurrency:
   Apps should be horizontally partitionable in a way that adding concurrency "is a simple and reliable operation", as vertical scaling is limited. Therefore the app has to be able to run processes that may be distributed amongst different machines.

9. Disposability:
   The processes mentioned before can be started and stopped without a large delay, making them disposable. This enables fast scaling as well as quick and easy deployment of updates. Another aim is to minimize startup time and build robustness against sudden deaths e.g. caused by hardware failures.

10. Dev/prod parity:
    Design apps for continuous deployment by minimizing the historical gaps between development and operation/production. This means to minimize the time gap between writing a code change and its actual release to production as well as involving the code-developers in the deployment process. Additionally the tooling in development and production should be as similar as possible to avoid unexpected errors. This urges the establishment of DevOps-Engineers that are both building and deploying/maintaining their software.

11. Logs:
    Logs give insights in the app's runtime behavior and should be treated as event streams with one event per line of the log. The app is not concerned with routing or storing its log stream and by default writes it to stdout or stderr. From there it can be captured and aggregated by another service running in the execution environment for long-time storage or analysis.

12. Admin processes:
    Administrational or management tasks should be treated as one-off processes and run in the same environment as regular app-processes. The necessary code should ship with the regular application code to avoid issues in synchronization. Thus, the Twelve-Factor-App favors languages that natively provide a Read-Eval-Print-Loop-Shell (REPL), making it easy to run one-off scripts.

For reference see [Wig17] and its sub-pages.

Enhancing the fourth factor (backing services), data management is a critical aspect of the microservice architecture. As data sources or other resources,

named backing services, should not be shared between microservices, decentralized data management has to be implemented. This way, each service handles its own database, separated from and inaccessible for other services. The decentralization of data management, introduces the possibility for Polyglot Persistence, an approach, where each service manages its own database, which can even be of an entirely different type than the databases of other services. For example, one service could use a MySQL-Database, while the next one uses NoSQL. In monoliths, usually transactions are used for multiple updates to guarantee consistency. But as this causes temporal coupling of services by combining their updates in transactions, it is not suitable for microservices. Thus, transactionless service-coordination is emphasized in microservice architectures and consistency is only guaranteed to be eventual consistency, that may be compensated using specifically implemented compensating operations. [FL14]

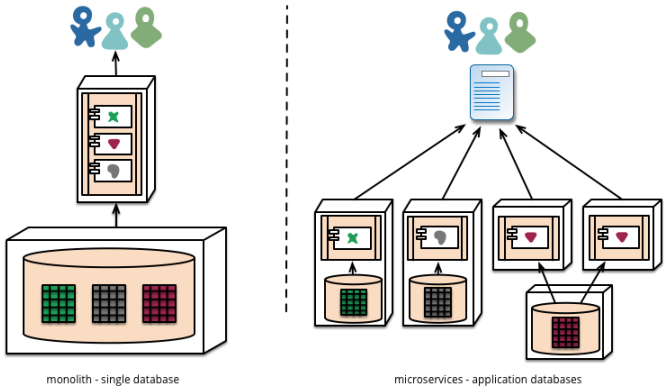

Figure 4: Decentralized Data Management [Fow]

Additionally, for microservices, the inter-service communication is a very important aspect that needs to be discussed in detail before implementation of a microservice architecture. Following Martin Fowler and James Lewis' principle of "smart endpoints and dumb pipes" [FL14], it is good practice to move smarts from the communication mechanisms, e.g. an Enterprise Service Bus, into the endpoints, making them as decoupled and cohesive as possible. [FL14] In general there are two approaches of designing inter-service communications:

1. **Synchronous Communication**
   The approach of synchronous communication mostly relies on choreographed messages using protocols like HTTP or lightweight messaging. It is a Request-Reply-Method, where a response is sent even in case of an error. While waiting for a response, the sender is blocked and can not execute any further actions. Methods include database-integration, REST (Representational State Transfer) as well as RPC-Methods like RMI (Remote Method Invocation), SOAP (Simple Object Access Protocol) and others. [FL14; Jan16]

   Example: Synchronous Communication Methods
   Re-using the Online-Shop example, purchasing an item from the shop requires placing an order using he checkout-service. The checkout-service therefore requests the customer-data from a customer-service. Now the communication between those two services happens via one of the three methods mentioned above this example:

- *database-integration*: the checkout-service skips the customer-service and directly queries the customer-database for the required data. This approach is quickly realizable and offers a good performance, but also servers as a typical anti-pattern, as it contradicts the principle of loose coupling, as it often requires proprietary database-drivers and might cause inconsistencies.
- *RPC*: the checkout-service calls a remote function like getCustomer() in customer-service. This approach as well is easily realizable and offers a defined schema for exchange but is afflicted by network-latencies. Additionally it relies on proprietary libraries and loose coupling might be limited by possible dependencies on other service's code.
- *REST*: the customer-service explicitly provides an API for requesting customer data which is queried by the checkout-service using an HTTP-GET-Request. This approach does not impose any additional need for coordination, as every request is complete. Also it is language-agnostic and produces human-readable output. On the downside, additional definitions of interfaces have to be created and the HTTP-protocol-overhead as well as possible latencies have to be handled.
  [Jan16]

2. **Asynchronous Communication**
   The "dumb" in "smart endpoints and dumb pipes" refers to the pipes not doing anything "smart" but only acting as a message router. Communication is achieved by sending messages over a lightweight message bus in a "fire-and-forget"-manner, where the message is sent, but the sender does not block to receive the response. That way, further actions can take place after sending a message, while request correlation, realized using e.g. GUIDs (Globally Unique Identifiers), makes responses comprehensible. That way, eventual consistency is achieved using transactions. Methods for using this approach include a scalable message-bus with a message-buffer. Tools like RabbitMQ or ZeroMQ offer these services while being lightweight and keeping the significant smarts in the endpoints. This supports scalability of a system, as, referring to the Online-Shop example from above, several customer-services could grab requests from the message-bus and reply to them, thus serving multiple requests at once. The message-bus also serves as the single spot for consistency-monitoring. Open-Source protocols like AMQP (Advanced Message Queuing Protocol) or STOMP (Simple/Streaming Text Orientated Messaging Protocol) can easily be used with those tools to achieve effective and scalable messaging and event-streams. [FL14; Jan16]

3.3 Refactoring a Monolith into Microservices

> "One of the biggest challenges is deciding how to split (partition) the system into micro-services."
> – Dmitry Namiot & Manfred Sneps-Sneppe, [NS14]

As already discussed in previous chapters, decomposing an existing legacy monolith into microservices can be very difficult, especially the initial process of defining the service boundaries.
Generally, refactoring a monolith into microservices means to decompose the monolithic application into components, separated in different packages, bundled with all their dependencies. Different modules inside each component have to belong to the same domain, maintaining high cohesion. As well, each component has its own database, relying on the best tool for the job and following the shared-nothing principle. [RG]

When moving an enterprise application from a monolithic to a microservices architecture, initially the organization itself has to undergo some structural changes.

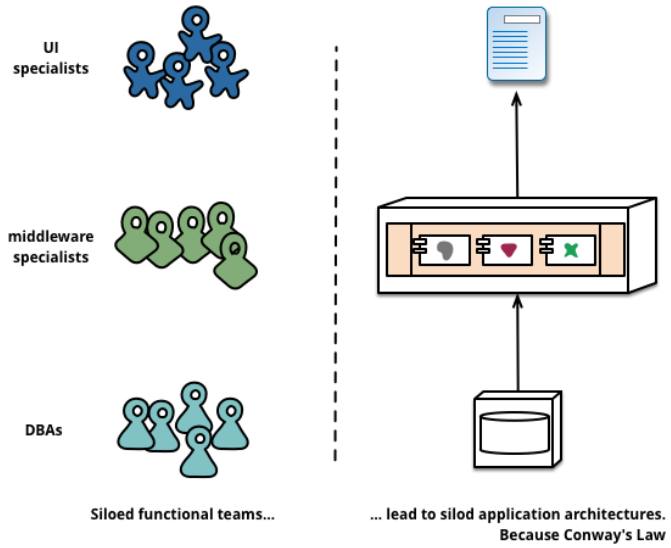

Figure 5: Conway's Law in action [FL14]

Traditionally, software development teams are organized along the technology layer, making up teams for User-Interface, server-management, databases, etc. The problem with this team structure is that "even simple changes can lead to a cross-team project taking time and budgetary approval" [FL14] and is therefore discouraged in a microservice environment. Instead the encouraged workflow is to "just force the logic into whichever application they have access to" [FL14], meaning that one cross-functional team should be responsible for every single aspect and every single piece of logic shaping their application. In the end this goes back to Conway's Law (see 2.3).

A common approach in this regard is to decompose an application into services that are organized around business capabilities. That way, also the developing teams are cross-functional and organized around the business capabilities rather than along the technology layer (see Figure 6 below). [FL14]

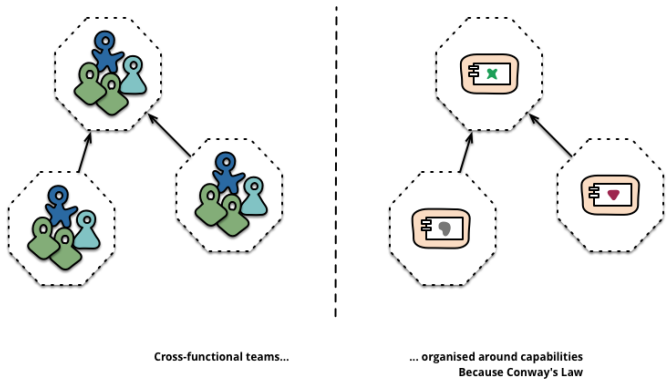

Figure 6: Service boundaries reinforced by team boundaries [FL14]

At the same time, a kind of DevOps-Culture has to be introduced to easily take care of lots of software artifacts developed in different languages on different platforms, etc. This will be discussed in the next section.

Before splitting up a monolith, it is advised to first think of what makes a good microservice. Qualities of a good microservice include context-separation via the shared-nothing principle, high cohesion via the single-responsibility principle, overall loose coupling and consistent communication via well-defined APIs. [Jan16] To achieve these qualities in microservices that replace a monolith, some hurdles have to be cleared. As the microservice architecture decentralizes almost every aspect of the application, also data management is decentralized. This leads to the need of eventual consistency. [Kra16] Still inconsistencies between decentralized databases have to be managed using compensating operations. [FL14] Often the most challenging hurdle that needs to be cleared lies in changing the communication pattern. In-Process communication has to be changed to Inter-Process communication. The issue with this is, that a "naive conversion from in-memory method calls to RPC leads to chatty communications which don't perform well" [FL14].

There are several approaches and patterns to split an existing monolith, of which most can as well be applied to greenfield-development of microservices. When splitting a monolith, one has to think extensively about where to place the cuts that separate the application into services. In terms of **evolutionary design**, the principles on which this can be decided are mostly based on the properties of the resulting components. Independent substitutability and updatability are key properties, which help setting the module boundaries in a way that rewriting a component would not affect its collaborators. Rarely changing parts of a system should be separated from the ones that change on a regular basis. Likewise, services that are repeatedly changed together because of the same reason should be merged into one, following the goal of high cohesion. [FL14]

While the properties are good guides to set module boundaries, more detailed design patterns exist:

- **Domain-Driven Design:**
  Domain-Driven Design (DDD) is sometimes referred to as the parent-pattern of some of the following design patterns, which inherit some approaches from it.

DDD describes a pattern for functional decomposition of applications, defined by Eric Evans. [RG] The general approach is to divide a complex domain up into multiple **bounded contexts**. Here the domain is the subject area to which a program is applied, while the context defines a sub-area of this domain, where a specific model applies. Eric Evans also defined the term of an "Ubiquitous Language", a language orbiting the domain, that can be used and understood by each member of a cross-functional team. [FL14; Eva03; Kra16]

- **Business-Driven Design:**
  Often, software is created to reproduce business processes for efficiency. Therefore, Business-Driven Design aims to keep code structure simple while projecting business processes onto code. To achieve this, the business domain is decomposed into separate and loosely coupled processes. In general, Business-Driven Design inherits most of its practices from Domain-Driven Design. The business-domain is split up into processes that belong to specific bounded contexts in this domain. Developers and domain experts identify those separate processes in the business workflow and label them with clear names, aiming for the ubiquitous language. As the business-domain defines the interfaces, the areas with the most interfaces between each other should be aggregated into contexts to maximize loose coupling. That said, a good context is one with the minimal possible number of interfaces from outside to the service. It is able to do many tasks on its own without consuming other interfaces and consumes data primarily from its own resources. Opposed to this, a bad context is in the worst case a context, where every action is an interaction and where data is consumed from legacy copies from other services.[Jan16; Kra16]

  > **Example: Contexts defined by number of interfaces**
  > There is an imaginative company featuring the departments Sales, Contracts, Payment and Finance. When analyzing the interactions amongst those areas, it becomes clear that there are a few interfaces between Sales/Payment and Finance/Contracts. On the other hand there are many interfaces between Sales/-Contracts and Payment/Finance. Thus, it would be the best approach to aggregate Sales/Contracts and Payment/Finance into two separate contexts for maximum loose coupling and high cohesion. [Jan16]

- **Customer-Driven Design:**
  In Customer-Driven Design, organizations try to replicate the customers' view on their application. Customers use different layers of a system (e.g. Shopping-Interface, Checkout-System, etc.) for different amounts of time (e.g. more browsing, less buying). Thus, different availabilities for the layers are needed. Therefore, scaling is dependent on customer needs, which is why maybe not every layer has to be optimized for scalability and e.g. Customer- and Checkout-Service could still remain in a monolithic core while the Shop-Interface-Service is excised into a microservice. Obviously this is not really a detailed design pattern but rather a more general approach to look on a monolithic application. [Jan16]

- **Team-Driven Design:**
  Using the Team-Driven approach, the monolith is decomposed into services according to the skills and capacities of the developing team. The size of a service is then defined by the domain-complexity correlated with the teams skills. For example it can easily get a team into trouble if they split a monolith into 15 microservice while only having around 10 developers, of which some are e.g. only specialized in front-end development. Service complexity can be rated by the criticality of the service and its expected amount of functions. [Jan16]

- **Technically-Driven Design:**
  Technically-Driven Design is kind of an anti-pattern that shows up amongst the approaches. It describes the separation of an application by technical aspects. E.g. a monolithic application is decomposed into microservices of which one is responsible for handling all the data (essentially a database-management-service) and another one handles the whole UI, etc. This is an anti-pattern as it introduces low cohesion and dense coupling between services. [Jan16]

- **Verb-Based Decomposition:**
  Decomposing an application by verbs or use-cases means to create services that are responsible only for particular actions, e.g. "a Shipping Service that's responsible for shipping complete orders" [Ric17b]. No other unrelated action should be implemented in this service.

- **Noun-Based Decomposition:**
  An application is decomposed "by nouns or resources by defining a service that is responsible for all operations on entities/resources of a given type, e.g. an account-service that is responsible for managing user accounts."[Ric17b] Thus, as opposed to verb-based decomposition, it might implement more than one action.

When planning to refactor a monolithic application into a microservice architecture, it is a good starting point to already have a component-oriented application. Else, there are two common approaches to start decomposition. On the one hand there is the approach of iterative decomposition, that starts with a 50/50-separation of the application. Then those parts are iteratively split again using 50/50-separation until the services are small enough. On the other hand, the approach of component-separation aims to separate components and interfaces efficiently by evaluating latencies and inter-component-dependencies. The general process of refactoring after analyzing the structure and dependencies is initiated by moving connected components into their own separate packages (e.g. Java-Packages). Then a dependency-graph can be created to draw out inter-package-dependencies, of which commonly used libraries can be extracted into a Utilities-Package. Based on the found interfaces (data-exchanges), an integration technology has to be found to integrate separated services into one system. Finally, the datasource has to be checked to e.g. find foreign keys in a database, in order to find suitable integration-interfaces for data-management. [Jan16]

3.4 Microservice Operations

With the introduction of microservices in an organization comes the need of also introducing a culture of automation, continuous integration and deployment as well as continuous adaption. With this, a DevOps-Culture (Development & Operations) becomes necessity. Martin Fowler emphasizes this by writing about "Products not Projects" [FL14], because for developers, an application (a microservice) should not only be a project that is handed over to a maintenance organization on completion. Rather the developers should own the full lifecycle of a product, from development over testing, deployment to maintenance and reacting to user-feedback. Amazon introduced the ethos "you build it, you run it" to emphasize this fact. [FL14] This DevOps-Culture gives the developers full responsibility for their software in production, where they have contact with its everyday behavior which also increases contact with users of the application. Definitely this is as well possible with monolithic applications, but the smaller the program, the easier it becomes to build up a personal relationship between developers and their users. [FL14]

As per above, continuous integration (CI) is a must in a microservice environment. CI is all about automating tests and automating build processes. Unit tests are executed immediately as well as integration tests that are executed as soon as possible. Those tests are combined in a fully automated build process. At the same time, code quality is checked and metrics regarding this quality are created and pipelined to the developer for assurance. [Jan16; HN17]
Tightly connected with CI and mostly talked about as one unit CI/CD is continuous deployment (or delivery). Assuming that all the tests succeed without an error, the build pipeline automatically deploys the application to various environments. In the end, the build pipeline automates the transitions between testing, maintenance and deployment thus massively reducing the time and effort put into running through this process manually as well as reducing the necessary coordination overhead between teams.[HN17; FL14]

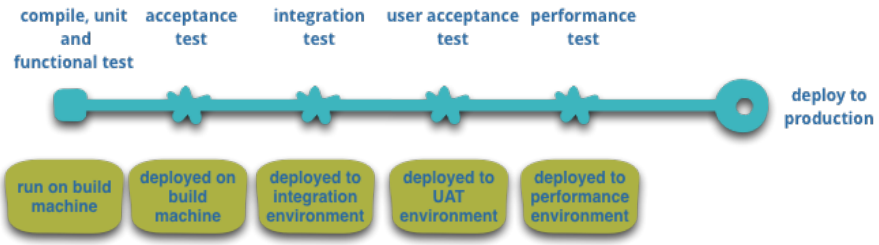

Figure 7: Basic Build Pipeline [FL14]

All in all, a fully automated CI/CD-Pipeline provides the "tooling for creating artifacts, managing codebases, standing up simple services or for adding standard monitoring and logging" [FL14]

The infrastructure often has to be adapted for the use of microservices. A hurdlefree infrastructure provisioning requires moving from a static server-structure (like it is traditionally established in many organizations) to a more flexible cloud infrastructure. The cloud system spans several servers and (virtual) machines and abstracts away any hard-

ware issues. There are also companies, which provide their cloud systems for external usage (e.g. AWS), also called Platform as a Service (PaaS). Applications deployed into the cloud can be distributed and scaled arbitrarily over the system of machines without "knowing" that they are not running on a single machine with other applications. [Jan16; FL14]

As many services run at the same time in the cloud without fixed addresses, a service discovery/registry is needed that identifies and tags the services in the cluster. This system can either run passively, as services register themselves with the service registry or the system can actively discover services e.g. by using tags. The registry is then queried by new clients to get the addresses of service-endpoints. Possible tools used for service registration and discovery include ZooKeeper, Etcd, Consul, Kubernetes, Netflix Eureka and others. At the same time, some service registries offer more functionalities like metadata-tagging, monitoring-binding and even Configuration Management Databases (CMDB) in which the current configurations of services are stored in a key-value database for service-configuration-correlation [Kra16; Jan16; RG]

Usually microservices are not deployed on a PaaS as packages or binaries but rather bundled up with all their dependencies in a container. Container architectures are themselves divided into two species. There are application-containers, offered by e.g. SpringBoot or NodeJS and system-containers by e.g. Docker or rkt. The most well-known container software is Docker, which is developed as open-source software, has a big community and offers clustering-mechanisms via e.g. Kubernetes or Docker-Swarm. [Jan16]

Often, there are several microservices running in parallel on a large cluster. As in some situations, service APIs should not be queried directly by a foreign client or e.g. there are endpoints of different versions available, an API-Gateway can be used to route messages/requests to the correct service. When forwarding a message to a service, the API-Gateway can even transform the message itself into a new format that is understandable by another version of the queried endpoint. The gateway itself queries the service registry to gather information about available endpoints and their versions. Examples of API-Gateways are the open-source projects Kong and Tyk, which also offer some security services and even endpoint simulations. [Jan16]

In a cluster of several machines, cluster-management and job-scheduling software handles the starting/stopping and distribution of applications across the available resources. Those tools, like Kubernetes, Nomad or Mesos with Marathon, take an application e.g. as a binary or a container along with a job-description and start/stop it as defined in the job-file. Considering the availability of resources, they automatically distribute the applications across the cluster to ensure maximum availability, resource usage and performance. Due to continuous surveillance of health statuses, they are able to automatically kill and restart unhealthy/dead/unresponsive jobs to maintain service availability. The same applies to automatic scaling based on service metrics. [Jan16] Using microservices it is a common mindset, that "software failure will occur, no matter how much and how hard you test" [RG]. Because of this fact and due to possible hardware failures it is necessary that the cluster-management software offers a failover protection that is able to restart the service somewhere else in the cluster, effectively hiding a service failure from the client. [FL14]

Due to the extreme need of real-time reactions and adaptions to software failures and workload changes in a microservice environment, it is necessary to provide effective logging and monitoring. Metrics enable the developing team to take proactive action if something goes not as planned in the production environment. Essential monitoring allows quick metric-based scaling and self-healing as described in the last paragraph as well as convenient planning for resource-capacity-enhancements. Metrics can be created via code-instrumenting from inside the application, based on container-/system-monitoring or by monitoring the communication channels. While this "semantic monitoring can provide an early warning system of something going wrong" [FL14], it can also create business relevant metrics like the number of orders or requests received per minute. Created metrics can be aggregated and visualized using tools like Prometheus or Graphite. Based on fetched metrics, these tools can also execute pre-defined alerting mechanisms e.g. to scale up an overloaded service. A uniform monitoring infrastructure across different teams is good practice as well as a centralized logging system. The often used ELK-Stack[4] (Elasticsearch, Logstash and Kibana) is used to aggregate, analyze and visualize distributed logs from different microservices. Log-Everything is an approach where every piece of provided data is written into a log to provide a solid base for functionality analysis of systems. In an environment of distributed microservices it is a requirement to use some log-integration system that correlates fetched log-messages with Globally Unique Identifiers (GUID) to make them traceable in database like Elasticsearch. A good logging system consists of several components, including the log-database (e.g. Elasticsearch), efficient aggregation, a forwarder (e.g. logstash), a buffer (e.g. Kafka or Redis) and a visualization and alerting software (e.g. Grafana or Kibana). [Jan16; FL14; RG]

[4]The ELK-Stack: https://logz.io/learn/complete-guide-elk-stack/

4 Case-study at trivago

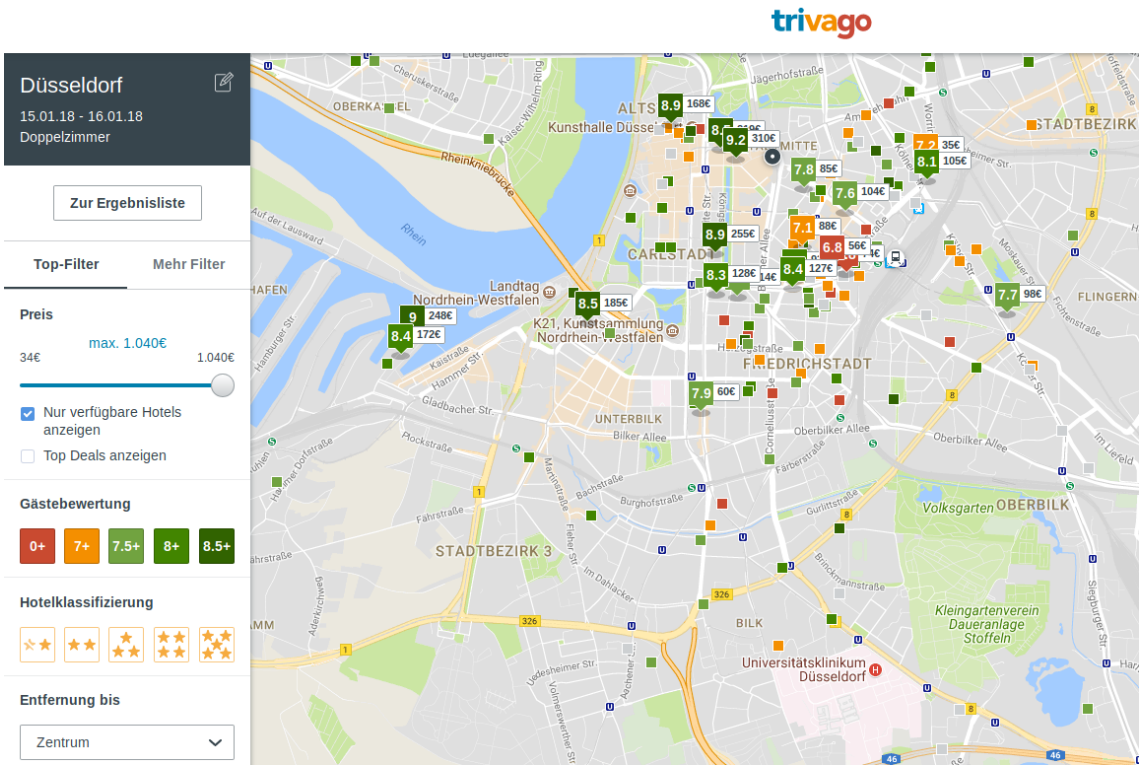

Figure 8: trivago's search results represented on a map

4.1 Structural Overview

trivago N.V. provides a metasearch engine to search for hotels world wide that are offered on different websites. To provide this service world-wide, trivago is hosted on several servers around the globe, owned by so-called Content-Delivery-Networks (CDN).
To ensure the website-performance and the efficiency of several internally developed tools, trivago hosts a website performance team as a sub-team of the software engineering department. The performance team is responsible to gather and analyze service and system metrics to find and repair bottlenecks as well as independently come up with performance improvements in used tools and workflows. Part of the work in the performance engineering team therefore includes monitoring the CDNs in order to get information about the reliability of their offered services and to gather performance metrics of the trivago web-service running on their remote servers. So far, all the metrics have been collected and processed by a single monitoring-monolith, written in PHP.

The set of CDNs includes ChinaCache and Edgecast CDN, which are (amongst other services) monitored using one big monolithic application in trivago's monitoring infrastructure. These providers offer HTTP-/REST-APIs, where different metrics from their platform can be fetched using simple HTTP-GET-Requests with specially formatted URL-

Parameters. Those metrics are read by the monolith and stored into a InfluxDB[5] time-series database using Graphite [6] as a middleware.

The objective of the implementation part of this thesis was, to split the existing monitoring-monolith up into several smaller microservices that are easier to maintain and handle. The following subsections will give a brief overview over the structure and problems of the existing monolith, why the team decided to move to microservices and how it was done. Next to the presentation of the general workflow, it will be explained which approaches/patterns from the previous chapters were applied and how they worked out.

During the flow of decisions and implementations, it is shown how two components from a big monolith into independent microservices were extracted (see highlighted parts in Figure 9).

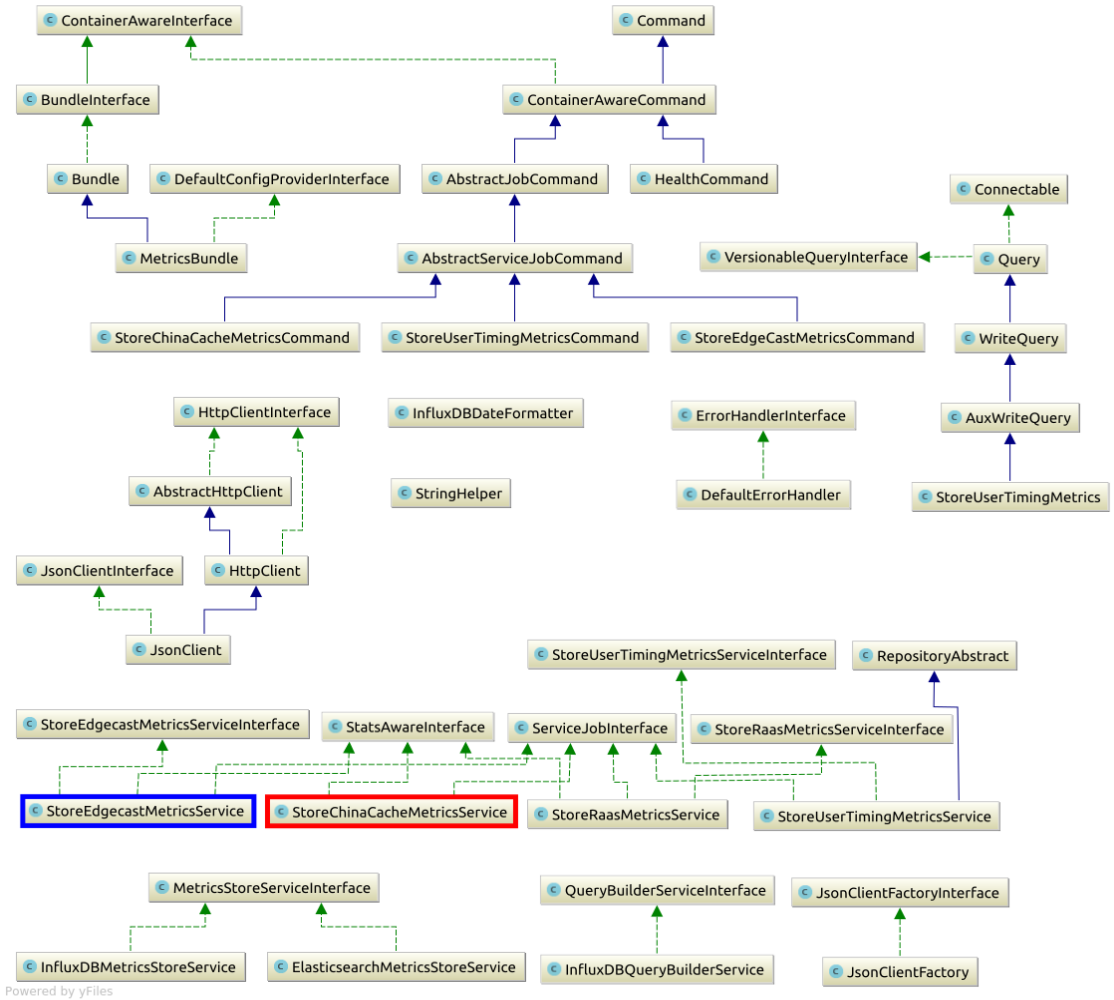

Figure 9: PHP Class Diagram of legacy monitoring-monolith with highlighted components for Edgecast and ChinaCache

[5]InfluxDB time-series platform: https://www.influxdata.com/
[6]Graphite is a monitoring tool used to store numeric time-series data: https://graphiteapp.org/

For new developers joining the project it will be easier to understand the small codebases (with flat hierarchies) of the new services than the big interwoven codebase (with a deep directory hierarchy) of the legacy monolith that also relies on some big frameworks.

4.2 Monolith at trivago

As mentioned in the last subsection, the legacy monolith is used for gathering metrics from CDN APIs and writing them to InfluxDB. In a broader view, it gathers metrics from several different sources and writes them to different storages. The monitoring-monolith basically wraps five different functionalities:

1. **CDN metrics → InfluxDB:** Queries the CDN APIs for metrics about website performance, bandwidth rate, etc. and writes the results to InfluxDB time-series database.
   (a) **Edgecast CDN API → InfluxDB**
   (b) **ChinaCache CDN API → InfluxDB**
2. **RaaS → InfluxDB:** Queries an internal service and writes the results to InfluxDB time-series database.
3. **ElasticSearch → Analytics DB:** Fetches metrics from an ElasticSearch-Cluster and writes the results to internal Analytics databases.
4. **User-Timing metrics → Analytics DB:** Fetches User-Timing metrics from InfluxDB and writes the results to internal analytics databases.

Currently, the monolithic application is built and included in a Docker[7]-container and deployed on a Mesos[8]-Cluster using Marathon[9]. On top of Mesos runs Chronos[10], a job scheduler. In this case, Chronos schedules a job for every component every five minutes. This job runs a copy of the Docker-container using different arguments for the starter scripts to execute the single components of the application. Thus, five containers featuring the same application are run at the same time, executing different components of the code. After returning from execution, the application and the container are killed and re-scheduled for the next run.

[7]Docker is a software for containerization of applications, making them infrastructure-agnostic: https://www.docker.com/
[8]Mesos is a distributed systems kernel: http://mesos.apache.org/
[9]Marathon is a container orchestration platform for Mesos:https://mesosphere.github.io/marathon/
[10]Chronos is a job-scheduler for Mesos: https://github.com/mesos/chronos

Focusing on the three components of the monolith that write to InfluxDB only, the workflow looks like this:

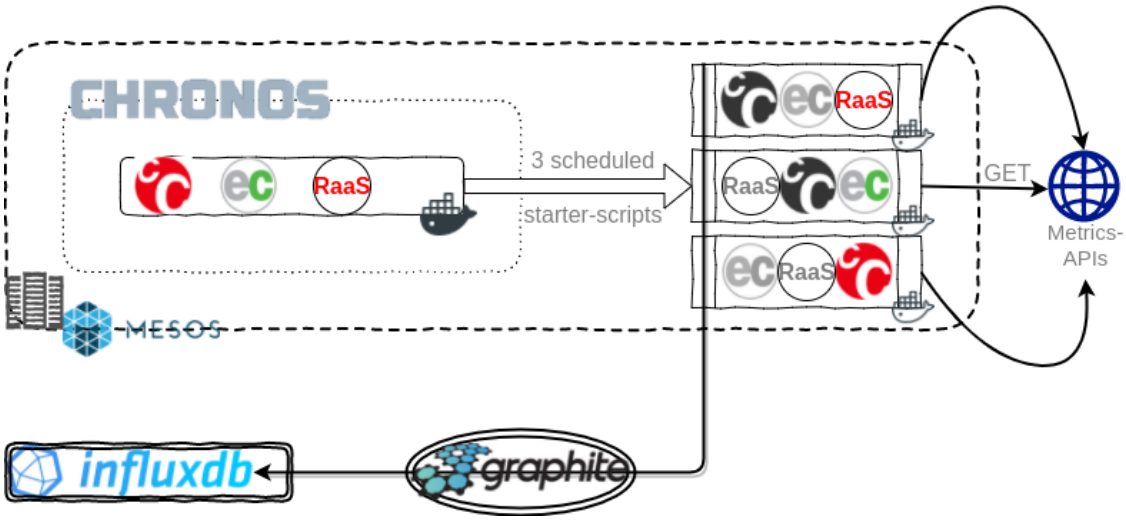

Figure 10: Workflow of legacy monolith, focusing components for Edgecast, ChinaCache and RaaS

In the course of this thesis, only the components described in 1.(a) and 1.(b) will be discussed. The components for Edgecast and ChinaCache work quite similarly:
Narrowed down to the essential steps, the components use given credentials to execute a HTTP-GET-Request against the respective API, requesting predefined metrics. The returned JSON-Response is then parsed into PHP objects. From those objects, the metrics value is extracted and sent to InfluxDB as a GaugeValue using Graphite.

The legacy monolith is written in PHP (Version 5) and makes extensive usage of big frameworks and platforms like Symfony [11] and Orchestra [12]. The strong dependence on frameworks and other helper-tools can be visualized in a dependency diagram, created using graph-composer[13]:

[11] Symfony is a set of reusable PHP components and a PHP framework for web projects: https://symfony.com/

[12] Orchestra Platform is an admin panel that handles Laravel extensions and user management: https://orchestraplatform.com/

[13] graph-composer: https://github.com/clue/graph-composer

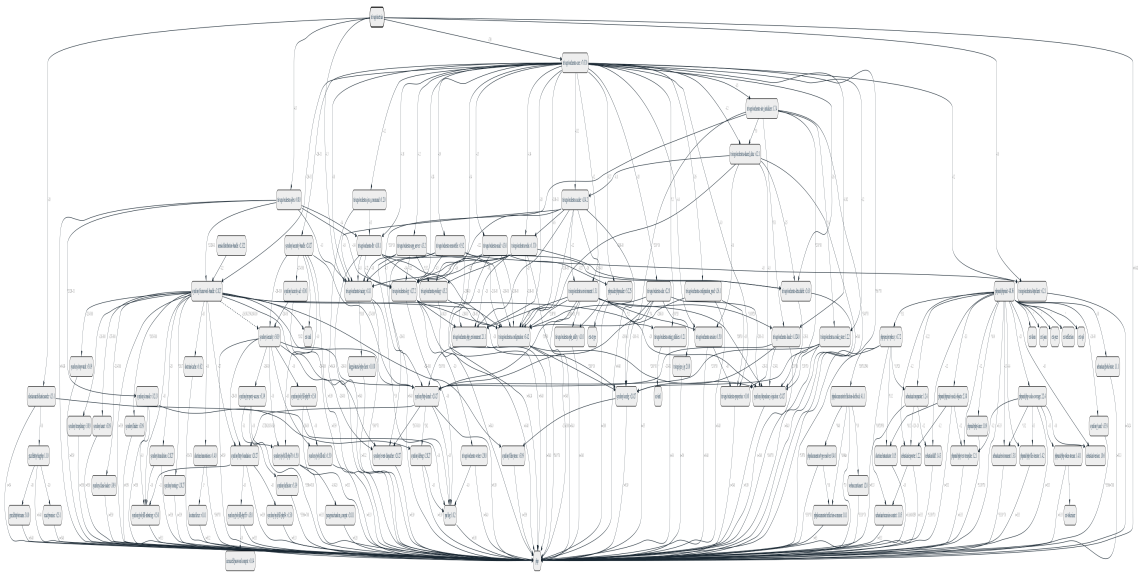

Figure 11: Dependency-Graph of monitoring-monolith, showing strong dependence on Symfony, Orchestra and other frameworks

As mentioned before, the monolithic application covers a big tech-stack including many different storage solutions, including InfluxDB, ElasticSearch, AnalyticsDB and uses a large tool-set, including PHP frameworks and Graphite.

The sheer size of the codebase as well as the interwoven dependencies make the maintenance of this application very difficult, especially when changes have to be done a few weeks or months after the last update and the responsible developer has to re-read the whole code to understand the functionality. Contrary to many legacy enterprise-size monoliths, this application is already built in a highly modularized manner with little inter-module dependencies. While the maintenance-aspect is the decisive reason for the refactoring-decision, there are as well some minor points in code-improvement that add to the sum. One of them is concurrency, as the legacy monolith simply executes the requests etc. one-by-one instead of running them all concurrently to speed up the process. Concurrency can speed up the request-process especially regarding the ChinaCache-Component, as response-latencies from the Chinese servers happen to be quite high sometimes. Additionally, new technologies for metrics-analysis and data storage are being introduced in the software engineering department and the application has to be adapted to them.

4.3 Refactoring Approach

Before starting to cut the legacy monolith into pieces, first a PHP Class Diagram of the whole `src/`-directory of the codebase was created, using JetBrains' PHPStorm[14] (see Figure 9). This was already useful to get a broader view over the module boundaries and inter-connections. Afterwards, graph-composer[15] was used to visualize the project's

[14]https://www.jetbrains.com/help/phpstorm/viewing-diagram.html
[15]https://github.com/clue/graph-composer

composer-file, to view its dependencies (see Figure 11). Due to the fact that the legacy monolith was already written in a very component-oriented way, it was quite easy to find tight module boundaries using the created diagrams.

Using Domain-Driven Design principles, especially focusing on bounded contexts, a clear image about the problem domain and its sub-domains could be drawn. The main problem-domain narrows down to "gathering and storing metrics", whereas its sub-domains include "metrics from CDNs to InfluxDB", "metrics from RaaS-Servers to InfluxDB", and more. Still, all those domains reside in the same codebase and separating them would again result in multi-functional applications responsible for several jobs rather than in microservices that are responsible for one job alone. That is the reason for analyzing the functionality of the monolith in-detail to find bounded contexts in the processes. The bounded contexts were good to find, as they were well-defined by metrics-suppliers. In the end, one bounded context equals to one job and one job is defined by "gathering metrics from one source (supplier) and write them to a specific data-store".

After clarifying the module boundaries and bounded contexts, trivago's performance engineering team aligned on two prioritized components that should be extracted first:

1. **Edgecast:**
   The Edgecast-Component is responsible for fetching and storing metrics from the Edgecast CDN API.

2. **ChinaCache:**
   The ChinaCache-Component is responsible for fetching and storing metrics from the ChinaCache CDN API.

The design-principles used to implement these two services should align with the ones mentioned in the previous chapters as good as possible, especially with the 12-Factor-App. As there were no communications between components in the monolith and the modules already worked on their own without depending on input from other components, there was no need to refactor the communicational patterns.

4.4 Preparation

Preceding any practical or theoretical preparations, the detailed workflow of the components, which should be refactored, was examined. To get a clear image of the functionalities, the plain source-code of the components analyzed first. Afterwards, the official API-documentations were read to analyze the HTTP-responses, which could then be compared to the output of the monitoring-monolith. After making notes about the flow of information through the components, the following task-description, which is quite similar for both components, could be created:

1. **Request-Preparation:** Use the configuration to build the Query-URLs used in the following HTTP-Request (i.e. paste credentials, requested metric, etc. in the preformatted API Endpoint URL)

2. **HTTP-Request:** execute a GET-Request against the prepared URL.

3. **Processing:** On successful response, parse the returned JSON into PHP objects, else retry or abort.
4. **Storing:** Extract the necessary metrics value from the object and use Graphite as a middleware to write it to InfluxDB as a GaugeValue.
5. **Repeat:** Repeat steps 1 to 4 for every considerable metric.

For the implementation of the new microservices, it was decided to use Go[16] as the programming language of choice for various reasons. As the main focus of the new services should be on maintainability, Go impresses with its ethos of "simplicity over elegance" which makes it a simple language that most programmers can pick up easily. Due to its enforced formatting via gofmt[17], the code has to be structured in a certain way, making the code look clean and easy to read. The big standard library as well as the garbage collection and the static type-system increase confidence in the Go codebase. Additionally, Go provides baked-in concurrency via goroutines[18] and a built-in HTTP- and JSON-Library, making it a good-to-use language for the desired services. Another strong argument for Go is that the codebases compile into single native binaries, making it platform-independent and easy to deploy. Therefore its "simplicity doesn't come at a price, it actually improves performance" and "eschews unnecessary complexity, ceremony and cruft"[HN17]. The many useful tools that ship with the Go installation make automation from the command-line even easier, making it a language and tooling, which fits perfectly to the concepts of a DevOps culture. [HN17; Bou17]

For writing the code, a JetBrains IDE named GoLand[19] was used, inter alia, because it offers support for FileWatchers (automatic execution of gofmt, goimports, gometalinter) and code completion, ensuring code quality.

After some discussions, trivago decided to abandon InfluxDB for this project and rather rely on Prometheus[20] as the only time series collection and processing server. Hence, also Graphite did not have to be used anymore. Prometheus was just introduced at trivago a few months earlier and is currently being established for monitoring purposes throughout the software development department. Prometheus is an open-source software written in Go and under active development, also supported by a large community and supplied with many integrations. It is also supported as a data-source by Grafana[21] for feature-rich visualization, which is already exensively used at trivago.

Following the examination of several frameworks and toolsets that support building microservices in Go, it was decided to design the service responsible for querying the Edgecast CDN API using the Onion-Model introduced by go-kit[22] (see [Bou17] for further information). The Onion-Model describes a way of separating code for logging and instrumentation from the actual (business-)logic. Regarding the Edgecast-Service, go-kit

[16] Golang: https://golang.org/
[17] Gofmt: https://golang.org/cmd/gofmt/
[18] Goroutines are lightweight threads managed by the Go runtime: https://tour.golang.org/concurrency/1
[19] GoLand: https://www.jetbrains.com/go/
[20] Prometheus is an open-source monitoring solution: https://prometheus.io/
[21] Grafana is used for time series analytics: https://grafana.com/
[22] go-kit on GitHub: https://github.com/go-kit/kit

was also a good choice because it supports code-instrumenting via Prometheus natively to also create application-metrics. The ChinaCache-Service on the other hand should refrain from such a specialty and use "traditional" logging in its code, especially for the purpose of comparison. Also, as this is not a critical service, logging and monitoring can be slightly neglected opposite to performance and readability.

To lock in and track dependencies, conforming the second factor of the 12-Factor-App, a dependency management tool had to be used from the beginning. Initially, glide[23] was used and later replaced by the officially supported dep[24] dependency manager which locks dependencies in a vendor-folder automatically, based on import statements in the code.

Gometalinter[25] is used for concurrently executing several static analysis tools for Go to ensure a good code quality.

4.5 Workflow

4.5.1 The Edgecast-Exporter

The practical part of this thesis started with extracting the component from the legacy monolith, that is responsible for fetching metrics from the Edgecast CDN API and writing them to InfluxDB using Graphite. Its coarse workflow is given in the previous chapter (see list on page 33 in section 4.4).

As an entry-point for the service implementation, an existing client for the Edgecast CDN API could be re-used that was previously developed by a team-colleague earlier this year but was unused at trivago until now. The Edgecast-Client[26], written in Go, provides the ability to query metrics regarding the bandwidth, HTTP-Statuscodes, connections and Cache-Statuses from the Edgecast CDN API for a provided Account-ID and Token. The results are then returned as usable Go-Structs instead of plain JSON:

```
1 // BandwidthData holds that data of a request
2 // to the edgecast bandwidth API
3 type BandwidthData struct {
4     Bps      float64
5     Platform int
6 }
```

Code 1: Excerpt from mre/edgecast/types.go representing a usable Go-Struct that is returned by the Edgecast-Client

As trivago decided on using Prometheus as the preferred server for collecting and processing time series data like metrics, the task could be narrowed down to writing a Prometheus-Exporter for the Edgecast CDN API. With the help of the Prometheus Documentation[27] and a talk by Brian Brazil at the PromCon 2016[28] the requirements were set.

[23] Masterminds' Glide: https://github.com/Masterminds/glide
[24] dep: https://github.com/golang/dep
[25] Gometalinter for static analysis: https://github.com/alecthomas/gometalinter
[26] Edgecast-Client by Matthias Endler: https://github.com/mre/edgecast
[27] Prometheus Documentation: https://prometheus.io/docs/instrumenting/writing_exporters/
[28] So You Want to Write an Exporter: https://www.youtube.com/watch?v=KXq5ibSj2qA

There were no known issues with latencies or failures with querying the Edgecast API, so a Prometheus-Collector was the most promising goal to aim for.

Sticking to the principles of the 12-Factor-App, every piece of configuration is gathered from the environment using Go's `os.Getenv()`-function. This includes the account-ID and token, which are needed to query the Edgecast CDN API for a specific account.

The microservice movement emphasizes service-monitoring and -logging. Hence using go-kit's Onion-Model promised a feature-rich integration of those two requirements. The code necessary for instrumenting and logging the functionality of the relevant functions of the service is outsourced into an `instrumenting.go` and a `logging.go` file. As the only functions that could possibly fail (e.g. due to network failures or the unavailability of the API-Service) are the ones that were already implemented in the external Edgecast-Client, an interface conforming the functions that should be monitored in the client had to be defined:

```
1 // EdgecastInterface to be used for logging and instrumenting-middleware
2 type EdgecastInterface interface {
3     Bandwidth(int) (*edgecast.BandwidthData, error)
4     Connections(int) (*edgecast.ConnectionData, error)
5     CacheStatus(int) (*edgecast.CacheStatusData, error)
6     StatusCodes(int) (*edgecast.StatusCodeData, error)
7 }
```

Code 2: Excerpt from trivago/exporter-edgecast/collector.go showing the interface used for the middleware

The logging-middleware then wraps each function defined in the interface by logging all of its data, including output and errors, while measuring the time the function took to return.

```
1  func (mw loggingMiddleware) Bandwidth(platform int)
2  (bandwidthData *ec.BandwidthData, err error) {
3      defer func(begin time.Time) {
4          _ = mw.logger.Log( // params: alternating key-value-key-value -...
5              "method", "Bandwidth",
6              "platform", fmt.Sprintf("%d(%s)", platform, Platforms[platform]),
7              "output", fmt.Sprintf("%+v", bandwidthData),
8              "err", err,
9              "took", time.Since(begin),
10         )
11     }(time.Now())
12     bandwidthData, err = mw.next.Bandwidth(platform) // hand call to service
13     return
14 }
```

Code 3: Excerpt from trivago/exporter-edgecast/logging.go showing the logging-middleware

In the same way, the other three functions defined in the interface are logged in detail.

The instrumenting-middleware works almost the same way, but instead of logging data-flows, it tracks the number of requests that were made and the time that was needed for a request to complete in order to give information about the service performance. To do so, the `main.main()`-function of the service declares the metrics that should be

monitored, in Prometheus format. Go-kit's implementation of Prometheus bundles a few steps for creating metrics into a single step and maps individual metrics to their Prometheus counterparts, thus saving some code. The interesting metrics for this service are the number of requests made per API-Metric and their latencies.

```
// Prometheus metrics settings for this service
fieldKeys := []string{"method", "error"} // label names
requestCount := kitprometheus.NewCounterFrom(prometheus.CounterOpts{
    Namespace: "Edgecast",
    Subsystem: "service_metrics",
    Name:      "request_count",
    Help:      "Number of requests received.",
}, fieldKeys)
```

Code 4: Excerpt from trivago/exporter-edgecast/main.go showing a tracked service metric

Again, this middleware is wrapped around the service and thus updates the metric-values for each instrumented function every time the respective function is called:

```
func (mw instrumentingMiddleware) Bandwidth(platform int)
(bandwidthData *edgecast.BandwidthData, err error) {
    defer func(begin time.Time) {
        lvs := []string{"method", "Bandwidth", "error",
            fmt.Sprint(err != nil)}
        mw.requestCount.With(lvs...).Add(1)
        mw.requestLatencyDistribution.With(lvs...)
            .Observe(time.Since(begin).Seconds())
        mw.requestLatency.With(lvs...).Set(time.Since(begin).Seconds())
    }(time.Now())
    bandwidthData, err = mw.next.Bandwidth(platform) // hand request to service
    return
}
```

Code 5: Excerpt from trivago/exporter-edgecast/instrumenting.go showing the updating of a service metric

The logging- and instrumenting-middleware is wrapped around the actual service-functionality by passing the service in as a struct-field for the middleware, which can then call the service's functions after applying its own logic:

```
// create EdgecastClient that communicates with the Edgecast API
var svc EdgecastInterface = edgecast.NewEdgecastClient(accountID, token)
svc = loggingMiddleware{logger, svc} // attach logger to service
```

Code 6: Excerpt from trivago/exporter-edgecast/main.go showing the wrapping of the logger around the service

The existing Prometheus Server can be configured to scrape predefined endpoints using their address. Therefore, to be scraped, the service has to expose its metrics on an HTTP-endpoint. In Go, this is easy to accomplish by registering the endpoint and letting it be handled by the `promhttp.Handler()` from the Prometheus package to ensure proper formatting and scrape-handling:

```
http.Handle("/metrics", promhttp.Handler()) // let promhttp handle requests
_ = logger.Log("err", http.ListenAndServe(":80", nil)) // start serving
```

Code 7: Excerpt from trivago/exporter-edgecast/main.go showing the HTTP-Handler

The collector, which utilizes the Edgecast-Client to gather the metrics from the Edgecast CDN API and then creates Prometheus Metrics from it, must implement the `Describe()`- and the `Collect()`-functions of the Collector-Interface defined by Prometheus. The `Describe()`-function has to pass the metric-descriptions of all metrics to a channel[29] that can be collected using this collector. The metric-descriptions for the Edgecast-Collector align with the metrics that can be fetched from the Edgecast CDN API and include an informative naming and help text:

```
bandwidth = prometheus.NewDesc(
    prometheus.BuildFQName(NAMESPACE, "metrics", "bandwidth_bps"),
    "Current amount of bandwidth usage per platform (bits per second).",
    []string{"platform"}, nil,
)
```

Code 8: Excerpt from trivago/exporter-edgecast/collector.go showing a metric-description

The `Collect()`-function on the other hand needs to gather all the metrics in the form of Prometheus-Metric-Objects and pass them to a channel to be collected. In this case, it iterates over the platforms that are available on Edgecast (`http\_small`, `http\_large` and others) and concurrently executes a `metrics()`-method for each one of them. This method then concurrently calls the methods `bandwidth()`, `connections()`, `statuscodes()` and `cachestatus()`, which then use the client to request the respective metrics from the remote API. Afterwards, they extract the metric values from the returned structs and create new Prometheus-Metrics, which are then passed into the provided channel:

```
ch <- prometheus.MustNewConstMetric(bandwidth, prometheus.GaugeValue,
    bwBps, []string{bwPlatform}...)
```

Code 9: Excerpt from trivago/exporter-edgecast/collector.go showing a new metric being passed into the channel

The concurrently running goroutines are synchronized using several instances of `sync.WaitGroup()` for every group of routines (grouped by metrics per platform).

The finished collector is created and registered with the Prometheus Registry in the main function:

```
collector := NewEdgecastCollector(&svc)
prometheus.MustRegister(collector)
```

Code 10: Excerpt from trivago/exporter-edgecast/collector.go showing the creation and registration of the collector

[29] Channels in Go: https://tour.golang.org/concurrency/2

The created metrics can be scraped by a Prometheus server or displayed in text format using a web-browser to visit the defined endpoint. To be scraped by a Prometheus server, the server's configuration-file has to include a scrape-configuration similar to this one:

```
scrape_configs:
    # scrape metrics exposed by the edgecast collector
    - job_name: 'edgecast_collector'
        scrape_interval: 5s
        metrics_path: /metrics
        static_configs:
            - targets: ['localhost:80']
                labels:
                    group: 'edgecast'
```

Code 11: Excerpt from prometheus.yml showing the scrape-job configuration for a Prometheus server running on the same machine as the service

In total, the finished Edgecast-Exporter works like this:

1. The service listens and serves on the specified endpoint using the `promhttp.Handler()`
2. On request (scrape), the collector uses the Edgecast-Client to query all the predefined metrics from the Edgecast CDN API

   - goroutines are used to concurrently run requests for every possible metric on every possible platform
3. The Edgecast-Client parses the resulting JSON-responses into Go-Structs and returns them to the collector
4. The collector reads the metrics from the struct fields and transforms them into Prometheus GaugeValues using the predefined descriptions
5. Those metrics are then passed through a channel to the collecting function and become readable on the defined endpoint

4.5.2 The ChinaCache-Exporter

The implementation of the ChinaCache-Exporter is based on the same workflow as the one from the Edgecast-Exporter with one exception: the metrics are not passively scraped from an endpoint by a Prometheus Server. Rather, the metrics are actively being pushed to an existing Prometheus Pushgateway[30], where they can be scraped by the Prometheus Server. This is justified by the fact, that the metrics from the ChinaCache CDN API might not instantly be available at the time of scraping, because experience showed, that latencies to the Chinese servers can be quite high and requests might even fail completely. As Prometheus could potentially discard metrics which were not scrapable for a fixed amount of time, it is safer to push them actively whenever they are available. Therefore, the main function of the ChinaCache-Exporter creates a new collector and pushes its metrics to a Pushgateway instead of having it listening on an endpoint:

[30] Pushgateway: https://github.com/prometheus/pushgateway

```
1 // create new client and hand it over to create a new collector
2 client := NewChinaCacheClient(user, pass, channelIDs, querytime)
3 collector := NewChinaCacheCollector(client)
4 err := push.AddCollectors( // push the metrics exposed by the collector
5     "ChinaCachePush",
6     nil,
7     pushGateway,
8     collector,
9 )
```

Code 12: Excerpt from trivago/exporter-chinacache/main.go showing the push-functionality

Additionally, there was no previously created client available for querying the ChinaCache CDN API. This client was created with similar workings and capabilities like the Edgecast-Client. Contrary to the Edgecast API, ChinaCache also requires configuration of Channel-IDs and a timeslot that should be queried. The metrics returned by the ChinaCache CDN API are also more detailed as they might contain information about single regions or ISPs (Internet Service Providers). Therefore, way more Prometheus Metrics are created by this service than by the Edgecast-Service.

In contrast to the Onion-Model used in the Edgecast-Exporter, the ChinaCache-Exporter does not make extensive usage of logging and instrumenting in the code, due to the fact, that the push-model in the implemented way requires the service to be restarted for every new query-round. Thus, service metrics would always be correlated with new and different instances of the same service. Additionally, the client, which covers the functionality that could supposedly be monitored for informations, is written as a module that could eventually be extracted into a stand-alone application, making the wrapping-approach pointless. Because of the low criticality of this service, neglecting logging and instrumenting for the sake of simplicity is acceptable. Still, essential logging in the service logic is done using the standard library.

4.6 Results

The resulting workflow of the two new services compared to the one of the legacy monolith (see Figure 10) might look a bit more complex (as explained in previous chapters, operations may get challenging with microservices), but in the end, it is just more fine-grained. The number of tools used to manage the services is the same as for the single monolith, but provides better control over every action, especially with the introduction of the service discovery/registry (in this case it is Consul[31]). The service discovery software also provides additional functionality like health checks with failure detection and configuration management databases.

[31] Consul for service discovery and configuration: https://www.consul.io/

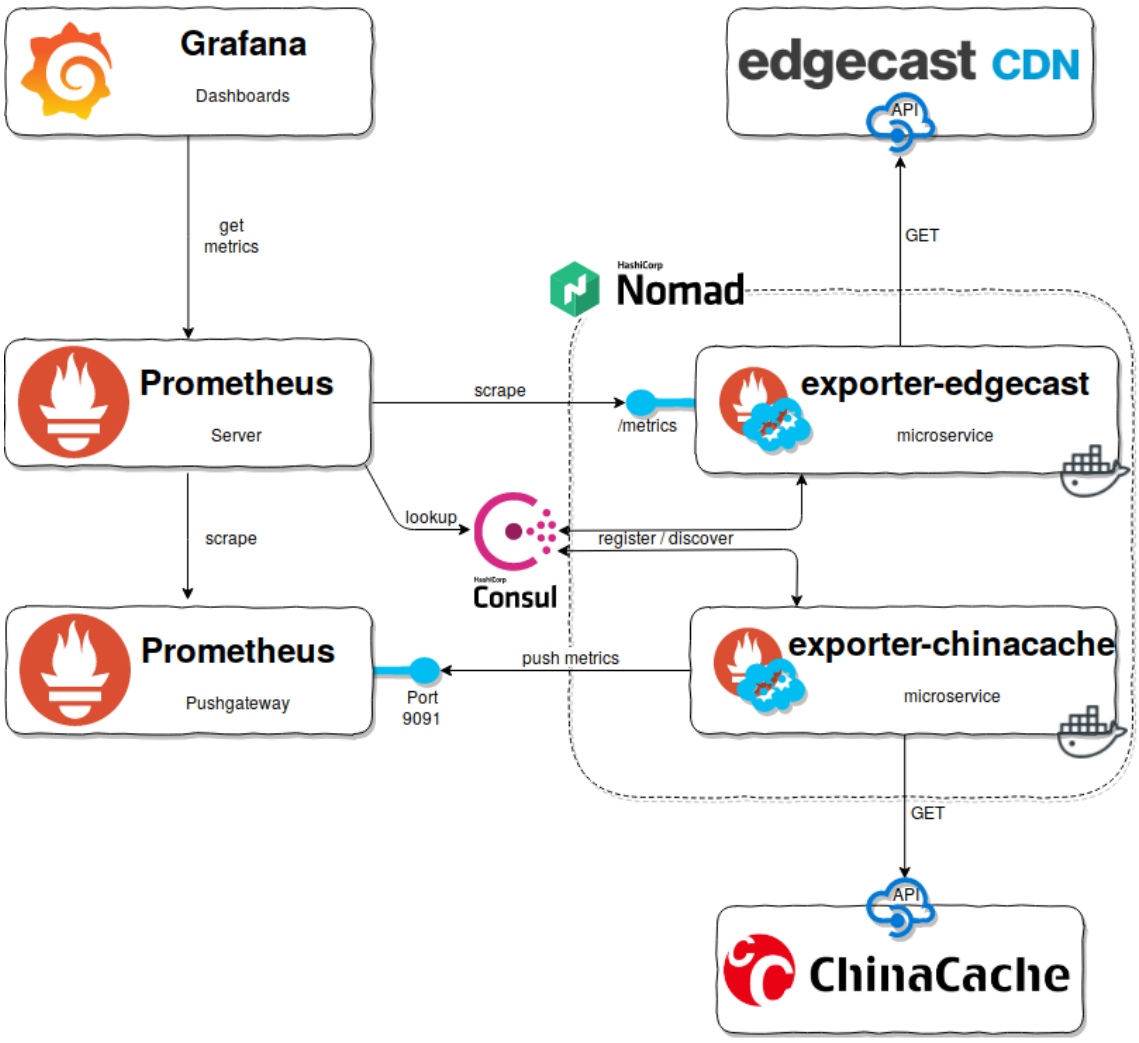

Figure 12: Combined workflow of both new services running inside individual Docker containers

Both microservices are wrapped in individual Docker containers, easily created with a simple Dockerfile with only two lines: `FROM golang:onbuild \\ EXPOSE 80`. Those containers are uploaded to Dockerhub[32] and their location is specified in the job file used for the cluster management tool Nomad[33]. The declarative job file also specifies the datacenter which the application should run on, port-mappings, naming and even tags for the service-discovery. In the case of the Edgecast-Service, the job-file includes (amongst others) the tag `trv-metrics` to let Consul know that this is a service that exposes metrics on the `/metrics`-endpoint. This way, the Prometheus Server can easily lookup the addresses and endpoints of every scrapable service on Nomad by simply querying Consul for services with the `trv-metrics`-tag. After scraping the metrics from the specified endpoints, Prometheus can display, analyze and visualize them itself. For even better visualization on large dashboards and for clean filtering, the metrics

[32]Docker Repository Service: https://hub.docker.com/
[33]Cluster magement and job scheduling: https://www.nomadproject.io/

source of the Prometheus Server is eventually consumed by Grafana.

The difference to this in case of the ChinaCache-Exporter is, that it does not provide the `trv-metrics`-tag, as it does not serve an endpoint, but rather actively pushes its metrics to the Pushgateway provided by the Prometheus Server. As the ChinaCache-Exporter should run periodically every few minutes and only push its metrics, when they're available, the Nomad job-file must specify the type `batch` and define a `periodic{}`-section that contains a definition similar to the one of cron-jobs under Linux.

On a side-note it is good to mention, that it is not mandatory to wrap the application in a Docker container, as Go compiles into single native binaries, which are as well supported by Nomad.

Those are some impressions from the usage in production:

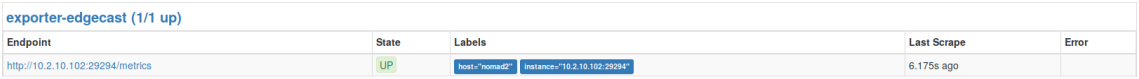

Figure 13: Edgecast-Exporter service shows up under /targets on Prometheus-Server

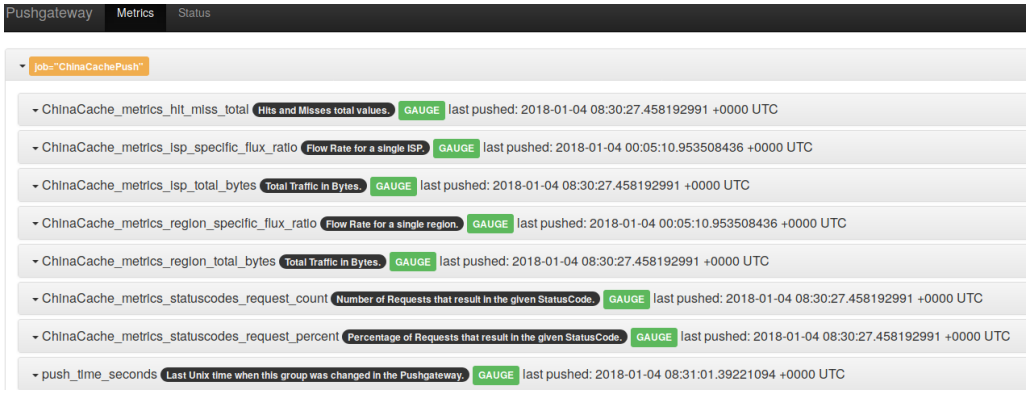

Figure 14: ChinaCache-Exporter shows up as a job in the Prometheus Pushgateway

```
# HELP ChinaCache_metrics_hit_miss_total Hits and Misses total values.
# TYPE ChinaCache_metrics_hit_miss_total gauge
ChinaCache_metrics_hit_miss_total{HitOrMiss="Hit",channel="[redacted],instance="",job="ChinaCachePush"} 6390
ChinaCache_metrics_hit_miss_total{HitOrMiss="Hit",channel="[redacted],instance="",job="ChinaCachePush"} 838
ChinaCache_metrics_hit_miss_total{HitOrMiss="Miss",channel="[redacted],instance="",job="ChinaCachePush"} 13
ChinaCache_metrics_hit_miss_total{HitOrMiss="Miss",channel="[redacted],instance="",job="ChinaCachePush"} 0
```

Figure 15: ChinaCache-Exporter metrics in text format on /metrics endpoint served by the Pushgateway

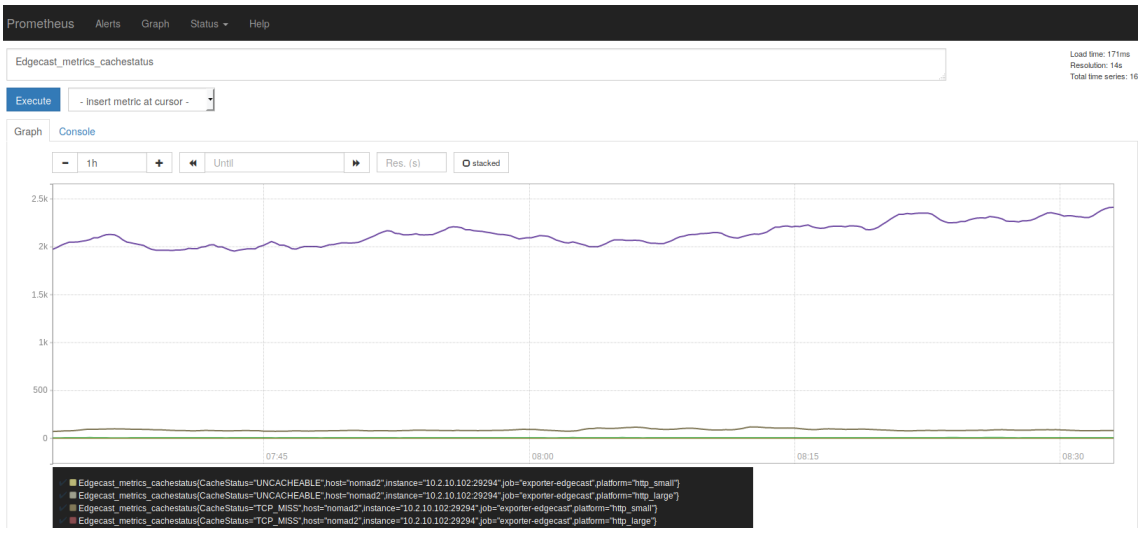

Figure 16: Edgecast bandwidth metrics graphical representation by Prometheus

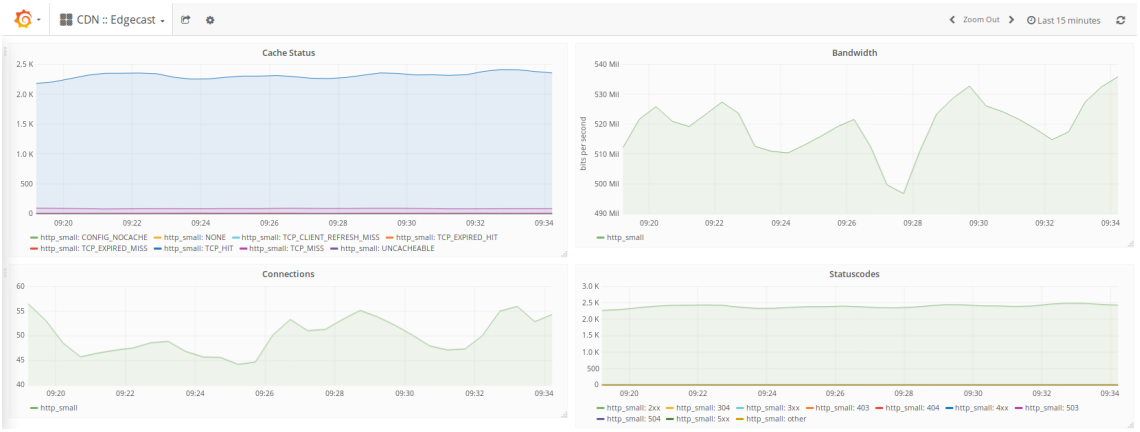

Figure 17: Dashboard in Grafana using Edgecast metrics provided by Prometheus

Information:
The source code of the described applications was released as open-source software under the trivago namespace on GitHub, using the Apache 2.0 license. The repositories can be found under the following links:

- https://github.com/trivago/exporter-edgecast
- https://github.com/trivago/exporter-chinacache

5 Evaluation

In a retrospective, extracting those two microservices from the legacy monolith was not too complicated, as the module boundaries in the monolith already were quite clear. As there were almost no inter-component connections at all in the old codebase, changing the internal communication pattern did not have to be considered, like it is often the case in enterprise-size applications that are refactored.

Looking at the old codebase compared to the two new codebases, some differences become quite obvious. While the amount of (self-written, not imported) code is almost equal, the new services are a bit more readable due to the componentization of the codebase and the enforced formatting in Go. More differences and their impact are given in the following table:

Results of the case-study for migrating a monolith into microservices at trivago

Legend: EC = Edgecast, CC = ChinaCache (referring to the respective services)

	Monolith	*Microservices*
Code-Base		
Modularization	- **no** - all the logic in a single file	- **yes** - logic semantically distributed over several files
Structure	- **nested** - many directories, deep hierarchy	- **flat** - a single directory, no hierarchy
Dependencies	- **many** - EC: Symfony Framework + Orchestra Platform - EC: 11 used imports - CC: Symfony Framework + Orchestra Platform - CC: 15 used imports (3 from standard library)	- **few** - EC: go-kit framework - EC: 12 used imports (6 from standard library) - CC: no framework/platform used - CC: 11 used imports (10 from standard library)
Functionality		
Concurrency	- **no** - API-Queries one by one	- **yes** - a goroutine for every API-Query
Logging	- **yes** - EC: simple logging, in line with service logic - CC: simple logging, in line with service logic	- **yes** - EC: extensive logging, Onion-Model via go-kit - CC: simple logging, in line with service logic

	Monolith	*Microservices*
Monitoring	- **no** - EC: no code instrumentation - CC: no code instrumentation	- **yes/no** - EC: instrumenting middleware, Onion-Model via go-kit - CC: no code instrumentation
Usage/Deployment		
Deliverables[34]	- **one big deliverable** - whole codebase in one container	- **several small deliverables** - single binaries (in containers)
Execution	- **complex** - Starter-Scripts: run.sh + bootstrap.php + others - Parallel execution of duplicated container	- **simple** - simple binary-execution - independent execution of binaries (in containers)
Configuration	- **static** - different configuration-files, stored with application	- **flexible** - flexible sets of environment-variables

Table 2: Case-Study Evaluation: Legacy Monolith (PHP) vs. New Microservices (Go)

The difference in the number of imported and used dependencies can be visualized using the following dependency graphs, generated using phpda[35] and dep[36]:

[34]Deliverables: https://en.wikipedia.org/wiki/Deliverable
[35]PhpDependencyAnalysis: https://github.com/mamuz/PhpDependencyAnalysis
[36]Visualizing dependencies with dep: https://github.com/golang/dep#visualizing-dependencies

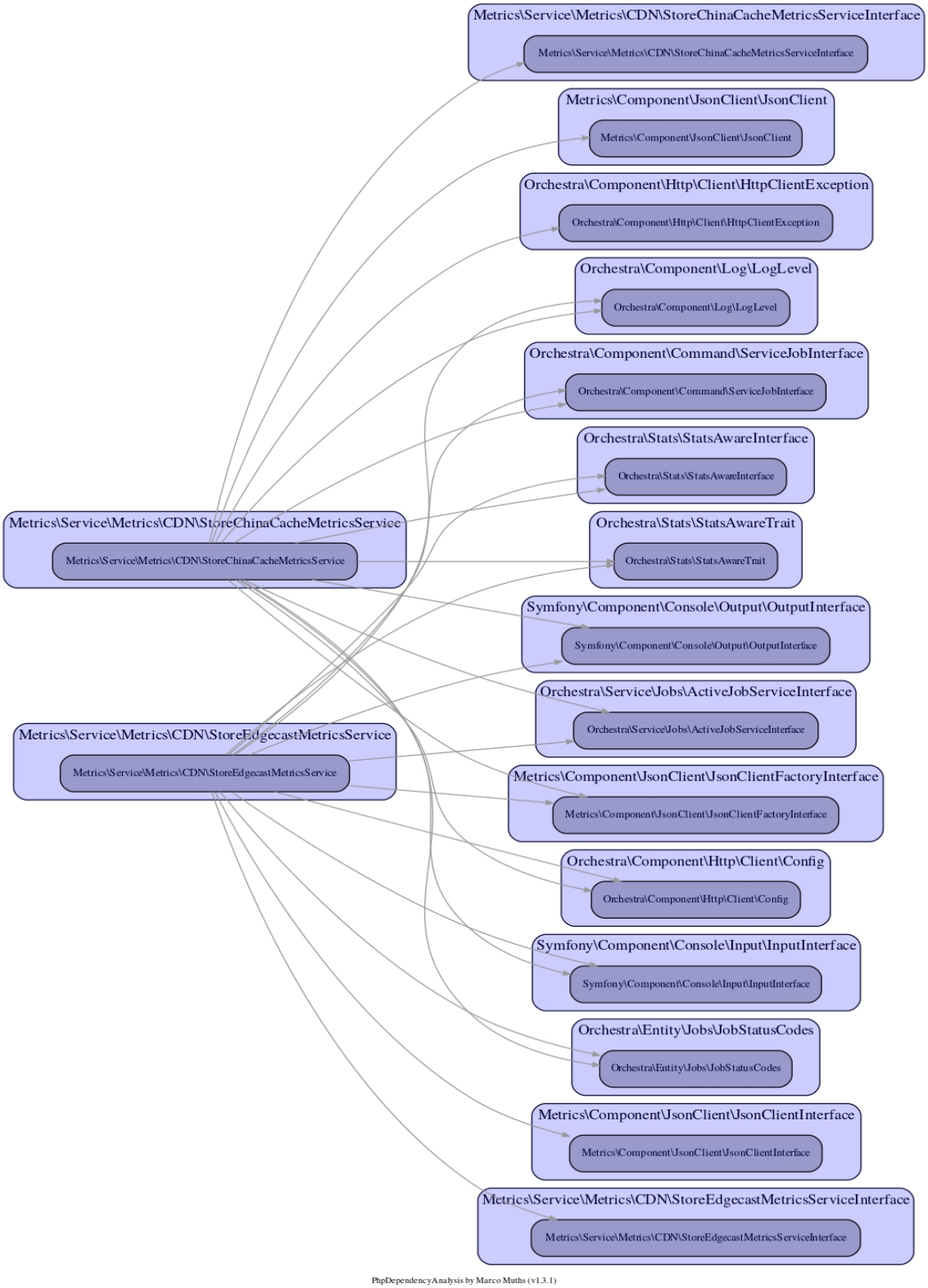

Figure 18: Usage (call + inheritance) Graph of monitoring-monolith shows strong dependence on big frameworks and platforms

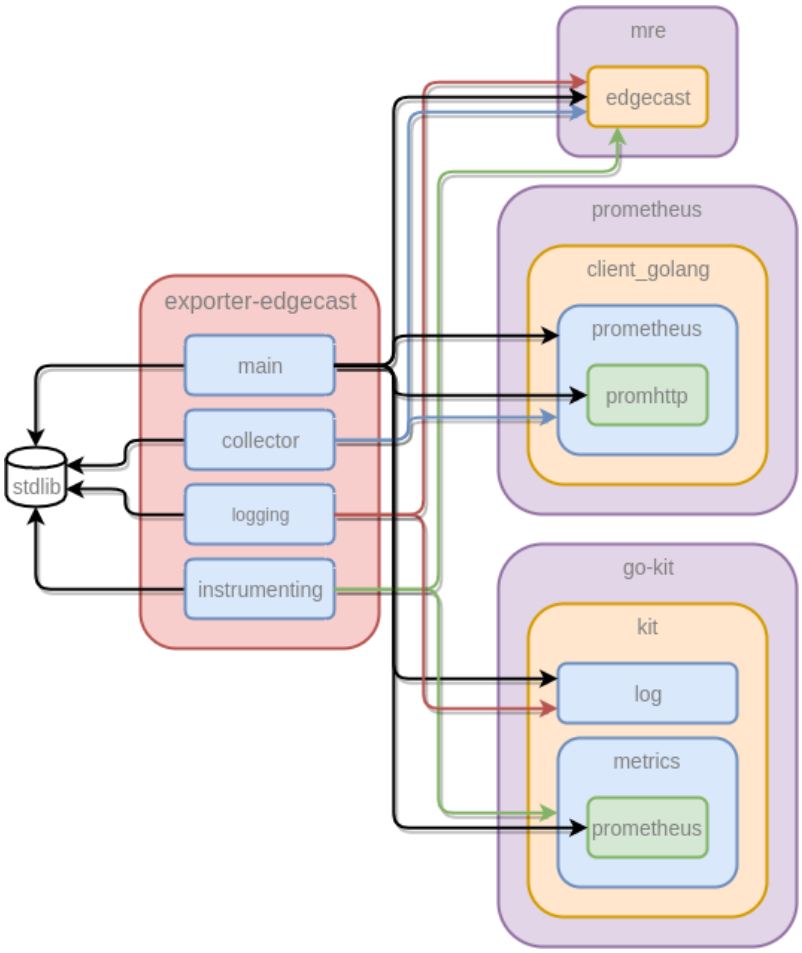

Figure 19: Dependency Graph of new Edgecast-Service shows dependence on some parts of a framework next to other external tools

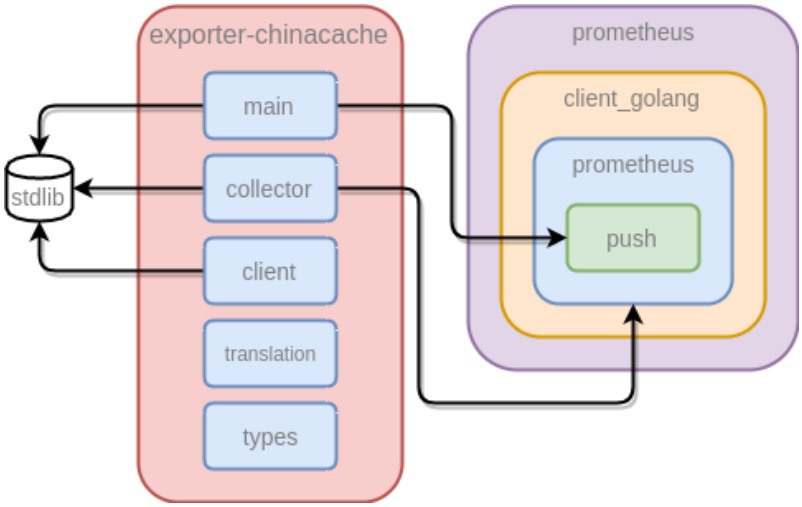

Figure 20: Dependency Graph of new ChinaCache-Service shows dependence on only one external tool

Comparing these graphs, it becomes clear that the new services use less internal and external dependencies, especially when comparing the ChinaCache service to the monitoring-monolith. Moreover, the new microservices do not heavily rely on big frameworks like the monitoring-monolith does.

All in all, the conclusion drawn from this brief comparison of critical aspects is, that despite reducing the overall complexity in the codebase, the new microservices introduce a set of new functionalities and more flexibility. This fact is in line with the goal of this refactoring approach, that was to reduce the complexity to improve maintainability of the monitoring tools.

6 Conclusion and Outlook

From the experiences of other people and companies with both microservices and monoliths that were analyzed for the theoretical part of this thesis as well as from the hands-on experience made during the practical part, still no sharp conclusion for general decision-making can be drawn. In contrast, the decision to further decompose the monitoring-monolith at trivago was already made, after reviewing the successful refactoring of the Edgecast- and ChinaCache-Services, which was presented within this thesis.

During the examination of several sources of information (textbooks, conference talks, etc.) for the theoretical part of this thesis, it became clear, that the microservice architecture tries to solve several problems of the monolithic architecture by emphasizing loose-coupling of services with high cohesion. But at the same time, the authors always emphasized, that every benefit of the microservice architectural style is a trade-off, a fact, that also showed up while refactoring the monolithic monitoring application at trivago. In a general view, e.g. the decentralization of data management allows for choosing the best database solution for each service individually but creates the challenge of distributed transactions or eventual consistency between them. In the case of the presented monitoring monolith, refactoring enabled the newly created services to use a completely different and up-to-date toolchain, while requiring to introduce and learn about new ways to manage them. That said, the new monitoring-microservices introduce a higher management-complexity at the time of creation and first usage, but they reduce the complexity of management and maintenance in the future.

At the same time, the comparison of the architectural styles in section 2.4 of this thesis as well as the evaluation clarified, that microservices are not the solution to every problem in software development. As discussed in more than one chapter of this thesis, the usefulness of this architecture rises with the complexity of the problem domain. For simple tasks with only very-few sub-processes or a domain with just a few bounded contexts, starting off with a microservice architecture would only introduce the additional problems that come with the development of distributed systems, while not reducing any complexity. In the case-study, Domain-Driven Design could easily be applied, considering the quite component-oriented design of the present monolith, which made it easier to find bounded contexts than it is often the case in enterprise-sized refactorings.

When it comes to refactoring an existing monolith into microservices, the same principle applies. If the complexity of the monolith is not high enough, refactoring it is not worth the effort. This especially applies for development and operational teams that would have to be reorganized in order to align to the cross-functional workflow that is required in a microservice environment.
Regarding the described case-study, the complexity of the monolith itself might not be high enough to justify the effort of refactoring. But despite that, the complexity of the code and the codebase is too high for easy and convenient maintenance. Additionally the legacy application missed some features and had to be updated in some places to get in line with the newly introduced tools like Prometheus.

In this bachelor thesis, the legacy monitoring application at trivago was partially decomposed into self-contained microservices. This seemed to be easy compared to the efforts of refactoring an enterprise-sized application, like Netflix did for a few years. Nonethe-

less, even this smaller-sized project with an application that was already built in a modular way, showed, that it requires considering many trade-offs as well as it requires extensive analysis of the monolith and its boundaries.

Talking about fast-paced development and updatability of software, the benefits of microservices opposite to monoliths or even SOA are pretty obvious. Being able to choose the best tool for the job for each service individually and the ability to individually update, replace, deploy and scale single microservices is a big benefit and can be an important advantage for organizations in market-competition. For the refactoring of the monitoring-monolith, choosing another language than PHP, which the monolith was written in, was a significant benefit. This is caused by the move to Prometheus as the sole time series processing and analyzing server. Prometheus and its set of tools are written in Go, thus their support and client library for Go provide easy integration and an impressive tooling. At the same time, the two extracted services required quick and easy HTTP-requests, where Go's `net/http`-package was very handy, which is already included in the standard library, thus reducing the need for external libraries. Despite that, upcoming extracted services might not require HTTP-integration but rather rely on a very good integration with ElasticSearch or a database, which might not have a full-fledged Go client library, thus making another programming language the best tool to choose for implementation.

While SOA and monoliths are currently still in-use in many (enterprise) applications, in the past few years, microservices seem to gain a lot of attention as being the architecture of choice for modern (web-)development. But in the end, even though microservices may solve several problems from traditional architectures, the costs that come with developing a distributed set of services should not be neglected. For every project, the trade-offs should be reflected extensively before deciding to use this uprising architecture. While the twelve-factor app aims to provide a guideline for optimal development of Software-as-a-Service, it also served as a stable base for the two new monitoring-microservices. Always checking back with the list, all the relevant aspects could be fulfilled. The codebases are tracked in revision control and a new deploy is made on every change. Dependencies are declared and isolated using dep as a dependency management software and every piece of configuration is taken from the environment. While the services are stateless and do not use any backing services for resources, at least the Edgecast-Service does export its service via port binding. Some other factors do not apply to the case-study, because of the smaller set of functionalities.

In the case of the examined monolith at trivago, refactoring the legacy monolith was worth the effort, as it introduced new features to the services and adapted them to new tools, which are now being used in the department, while also reducing the code-complexity, thus simplifying maintainability.

Looking back at the successful extraction of the Edgecast- and ChinaCache-Services in the form of Prometheus-Exporters, the same patterns and approaches can be applied to refactor the `StoreRaasMetricsService` (see class next to the red-marked class in figure 9). Because of the already proven easier readability and maintainability of the new services along with the usage of a new tooling in the monitoring infrastructure, all the separate components of the legacy monitoring-monolith shall be extracted in the future. After extracting the RaaS-Service, new approaches have to be examined to find perfect

solutions that integrate with e.g. the Analytics database or ElasticSearch rather than with a HTTP-API to successfully decompose the remaining monolith.

References

[Bar17] Douglas K. Barry. *Service-Oriented Architecture (SOA) Definition*. 2017. URL: https://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html (visited on 12/17/2017).

[Bou17] Peter Bourgon. "Go + Microservices = Go Kit [I] - Peter Bourgon, Go Kit". Cloud Native Con - Europe 2017. Europe, 2017. URL: https://www.youtube.com/watch?v=NX0sHF8ZZgw (visited on 12/16/2017).

[Cha17] Mike Chan. *Microservices vs. Monoliths: What's the Right Architecture for your Software?* Thorn Technologies. Dec. 13, 2017. URL: https://www.thorntech.com/2017/12/microservices-vs-monoliths-whats-right-architecture-software/ (visited on 01/12/2018).

[Con68] Melvin E. Conway. *Committees Paper*. 1968. URL: http://www.melconway.com/Home/Committees_Paper.html (visited on 12/18/2017).

[Dic] Dictionary. *Monolith | Define Monolith at Dictionary.com*. URL: http://www.dictionary.com/browse/monolith (visited on 12/17/2017).

[EI16] Josh Evans and InfoQ. "Mastering Chaos - A Netflix Guide to Microservices". QCon - San Francisco 2016. San Francisco, 2016. URL: https://www.youtube.com/watch?v=CZ3wIuvmHeM (visited on 12/16/2017).

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1st ed. Addison-Wesley Professional, Aug. 22, 2003. 563 pp.

[FL14] Martin Fowler and James Lewis. *Microservices*. martinfowler.com. Mar. 25, 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 12/16/2017).

[Fow] Martin Fowler. *decentralised-data.png*. URL: https://martinfowler.com/articles/microservices/images/decentralised-data.png (visited on 12/27/2017).

[Fow14a] Martin Fowler. "GOTO 2014 • Microservices • Martin Fowler". GOTO Conference - Berlin 2014. Berlin, 2014. URL: https://www.youtube.com/watch?v=wgdBVIX9ifA (visited on 12/16/2017).

[Fow14b] Martin Fowler. *Microservices Guide*. Microservices Resource Guide. 2014. URL: https://martinfowler.com/microservices/ (visited on 12/16/2017).

[Fow15a] Martin Fowler. *Microservice Trade-Offs*. martinfowler.com. July 1, 2015. URL: https://martinfowler.com/articles/microservice-trade-offs.html (visited on 12/16/2017).

[Fow15b] Martin Fowler. *MicroservicePremium*. martinfowler.com. May 13, 2015. URL: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 12/16/2017).

[Fow15c] Martin Fowler. *MonolithFirst*. martinfowler.com. June 3, 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html (visited on 12/16/2017).

[Ger16] Sanchit Gera. *An Introduction to Microservices*. freeCodeCamp. Apr. 4, 2016. URL: https://medium.freecodecamp.org/an-introduction-to-microservices-2705e7758f9 (visited on 12/16/2017).

[HN17] Kevin Hoffman and Dan Nemeth. *Cloud Native Go: Building Web Applications and Microservices for the Cloud with Go and React*. 1 edition. Hoboken, NJ: Addison-Wesley Professional, Jan. 2, 2017. 256 pp. ISBN: 978-0-672-33779-6.

[Jan16] Christopher Janietz. "Microservices als Architekturmuster". Video-Training. LinkedIn Learning Online. May 3, 2016. URL: https://www.video2brain.com/de/videotraining/microservices-als-architekturmuster (visited on 12/16/2017).

[Kra16] Filip Krakowski. "Architekturexperimente mit Microservices". Bachelorarbeit. Düsseldorf: Heinrich-Heine Universität, Sept. 29, 2016.

[Mes16] Ruslan Meshenberg. "GOTO 2016 • Microservices at Netflix Scale: Principles, Tradeoffs & Lessons Learned • R. Meshenberg". GOTO Conference - Amsterdam 2016. Amsterdam, 2016. URL: https://www.youtube.com/watch?v=57UK46qfBLY (visited on 12/16/2017).

[Net17] Netflix. *SimianArmy: Tools for keeping your cloud operating in top form. Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures*. original-date: 2012-07-06T22:00:05Z. Dec. 24, 2017. URL: https://github.com/Netflix/SimianArmy (visited on 12/24/2017).

[New15] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. 1 edition. Beijing Sebastopol, CA: O'Reilly Media, Feb. 20, 2015. 280 pp. ISBN: 978-1-4919-5035-7.

[NS14] Dmitry Namiot and Manfred Sneps-Sneppe. "On Micro-services Architecture". In: *International Journal of Open Information Technologies* 2.9 (Aug. 30, 2014), pp. 24–27. ISSN: 2307-8162. URL: http://injoit.org/index.php/j1/article/view/139 (visited on 12/16/2017).

[Ran16] Matt Ranney. "GOTO 2016 • What I Wish I Had Known Before Scaling Uber to 1000 Services • Matt Ranney". GOTO Conference - Chicago 2016. Chicago, 2016. URL: https://www.youtube.com/watch?v=kb-m2fasdDY (visited on 12/16/2017).

[RG] Vineet Reynolds and Arun Gupta. *Getting Started With Microservices - DZone - Refcardz*. dzone.com. URL: https://dzone.com/refcardz/getting-started-with-microservices (visited on 12/16/2017).

[Ric17a] Chris Richardson. *Applying the microservice architecture pattern language*. 2017. URL: http://microservices.io/articles/applying.html (visited on 12/24/2017).

[Ric17b] Chris Richardson. *Microservices Pattern: Microservice Architecture pattern*. microservices.io. 2017. URL: http://microservices.io/patterns/microservices.html (visited on 12/18/2017).

[Ric17c] Chris Richardson. *The Scale Cube*. 2017. URL: http://microservices.io/articles/scalecube.html (visited on 12/19/2017).

[Sal] Heroku Salesforce. *The Heroku Platform as a Service & Data Services | Heroku*. URL: https://www.heroku.com/platform (visited on 12/24/2017).

[Sch17] Hartmut Schlosser. *Technology trends 2017: These are the top architecture trends*. JAXenter. Mar. 22, 2017. URL: https://jaxenter.com/technology-trends-2017-top-architecture-trends-132550.html (visited on 01/12/2018).

[SW04] David Sprott and Lawrence Wilkes. *Understanding Service-Oriented Architecture*. Jan. 1, 2004. URL: https://msdn.microsoft.com/en-us/library/aa480021.aspx#aj1soa_topic2 (visited on 12/17/2017).

[Thö15] J. Thönes. "Microservices". In: *IEEE Software* 32.1 (Jan. 2015), pp. 113–116. ISSN: 0740-7459. DOI: 10.1109/MS.2015.11. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7030212 (visited on 11/13/2017).

[Til15] Stefan Tilkov. *Don't start with a monolith*. martinfowler.com. June 9, 2015. URL: https://martinfowler.com/articles/dont-start-monolith.html (visited on 12/20/2017).

[Web] Merriam Webster. *Definition of MONOLITH*. URL: https://www.merriam-webster.com/dictionary/monolith (visited on 12/17/2017).

[Wig17] Adam Wiggins. *The Twelve-Factor App*. The Twelve-Factor App. 2017. URL: https://12factor.net/ (visited on 12/16/2017).

List of Figures

List of Tables

Listings