Kafkalytics: Anti-pattern detection in a Kafka message bus

Master's Thesis

Rogier Slag

Kafkalytics: Anti-pattern detection in a Kafka message bus

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rogier Slag
born in Eindhoven, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Kafkalytics: Anti-pattern detection in a Kafka message bus

Author: Rogier Slag
Student id: 1507761
Email: R.G.J.Slag@student.tudelft.nl

Abstract

Recently microservices have emerged as a new architectural pattern which promises many advantages. Services are modeled along business entities, which should result in a flexible system. Apart from that the pattern promises better fault resilience against outages and better performance regarding scalability.

In this paper we explore the differences between several architectural styles where we focus on microservices. Additionally we discuss the actual usage of the architures in practice, based on interviews with industry experts. Talks with these experts indicated several problems regarding communication between services.

We conclude by identifying several antipatterns when using a Kafka message bus and present a tool Kafkalytics which can detect these patterns. Subsequently Kafkalytics has been implemented in a live production environment where it was evaluated as a tool with low overhead which is able to detect various antipatterns in a live production environment. It allows to be gradually implemented in a complete system and leverages existing infrastructure systems.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member: Dr. P. Pawelczak, Faculty EEMCS, TU Delft

Contents

Chapter 1

Introduction

1.1 Context

A few years ago James Lewis and Martin Fowler popularized the term microservices [James Lewis, 2014]. They described an architecture which aims at decoupling of software components by creating separate small services around business capabilities. Although already in place at some organizations (sometimes as Service Oriented Architectures), the concept was not widely known yet. Over time microservices have become increasingly popular as a way of structuring applications.

When I worked at a startup, we also decided to adopt this pattern. In order to prevent coupling between several services, we placed a lot of emphasis on the decoupling of services. The resulting architecture therefore governed that services should work together, without being coupled to each other.

As a part of the solution we determined that a message bus would be central to the system (in our case Apache Kafka, a distributed message bus system which can be found at https://kafka.apache.org/). On this bus each service could publish actions which it deemed of importance. Other services where then able to listen for these messages, and determine how to respond to each message (either by handling or discarding the message). Using this approach the team was able to create a strongly decoupled architecture of independent services. Together this set of services fulfilled the business requirements of the system as a whole. An example system is shown in Appendix A, which is based on this production system.

The resulting system was deemed a success, and is still running in production at the time of writing. However we also discovered along the way a number of best and worst practices was sometimes missing. In order to speed up the adoption rate of the microservice architectural patterns, this needed to be addressed.

1.2 Problem statement

Despite the current popularity of microservices, there is little known to date on the pros and cons of these systems in the long run. This popularity may turn out to be similar to the situation in the 2000s, when Service Oriented Architectures were a hot topic. As we will describe in Chapter 4 this style also turned out to have its problems. Whereas microservices seem to quickly deliver some of their promises, currently less is known on the long term maintenance of such system. How these systems tend to

evolve over time is therefore still an open research question. Problems may arise in communication between services, the detection of unused code, or other areas.

When dealing with microservices, several communication methods can be employed (which will be described in Section 2.2.2). One of those is a message bus system, where services pass information by publishing messages to a central bus and subscribe to certain types of messages. In an organization where every service is free to access this bus, define new messages, and read any message, without central governance this may seem like a disaster waiting to happen. Currently there seems to be a lack of knowledge in this regard.

The goals of this thesis are therefore

- To obtain a deep understanding on how microservices are used in practice,
- To gain knowledge on the different types of communication in these systems,
- To identify possible antipatterns when using a non-governed bus system,
- To provide a working solution which detects antipatterns when using such a bus in a microservice architecture in the production environment of an organization.

1.3 Approach

To archive the goals stated above, we decided to conduct interviews with industry experts. These experts all come from the IT industry itself (no academia), and were therefore able to give a first hand account on how software is used within their organizations. Each of these organizations develops software, either for customers or for internal use. The setup for the interviews will be described in Chapter 3.

The first set of interviews revolved around the use of monolithic and service oriented architectures in these organizations. The advantages and disadvantages as experienced by these organizations were discussed along with any problems the organizations faced when using these approaches. With the organizations which also employed a microservice architecture, an additional discussion took place. This discussion also focussed on reasons why the architecture was chosen and how this style influenced the development practice.

These interviews confirmed the initial conjecture that a number of antipatterns may occur when creating the communication between the services. Moreover the participants expressed they did not have a clear understanding of how technical debt may arise in their setups, or how to avoid this process. They indicated that would welcome a tool which was able to inspect their environments automatically with just little overhead. This tool should be able to pinpoint possible problems or antipatterns.

Therefore we identified a number of antipatters regarding communication between services. We focus on the message bus system Kafka, however these antipattern may also be valid for other message bus systems. These antipatterns were discussed with the earlier experts to determine to which extent these patterns were present in their designs. When confronted with the list of antipatterns and their descriptions these experts also indicated to which extent each antipattern could lead to problems in their systems.

Finally we provide a concrete tool (dubbed Kafkalytics) which is able to detect these antipatterns in live production environments. This tool focuses on easy access to data, and tries to minimize operational overhead and implementation effort. The tool was subsequently implemented by one of the companies, where it analyzed the message bus data for 2 months. After these two months antipatterns were successfully highlighted by Kafkalytics, after which the organization was able to repair these issues.

We evaluate Kafkalytics by

- Its capability to adequately detect antipatterns,
- The implementation overhead introduced by the system when the system is added to existing services,
- The operational overhead of adding Kafkalytics to a production environment.

1.4 Thesis overview

A background on microservices is given in Chapter 2. This background is intended to familiarize readers not entirely familiar with the concept in order to give them an adequate insight in the architecture. In Chapter 3 we discuss the interview phase, its setup and the selection of the participating organizations.

Chapter 4 takes the participants from Chapter 3 and reflects with them on two other dominant development styles: monolithic and Service Oriented Architectures. The participants also reflect on how these styles are used in their organizations, and discuss any experienced advantages and disadvantages. Chapter 5 reflects on how microservices are used in practice by the organizations introduced in Chapter 3. It also highlights differences between this style, and the styles of Chapter 4.

Kafka and Kafkalytics are introduced in Chapter 6. Specifically Section 6.1 will familiarize the reader with the core concepts of Kafka itself. The design paradigms of Kafka are also compared to that of the ESB (Enterprise Service Bus) which was popular in SOA setups. Section 6.2 highlights some antipatterns which may be present when using a message bus system like Kafka. Finally Section 6.3 explains the goals of Kafkalytics and list the requirements for evaluation. Section 6.4 indicates how Kafkalytics is be able to detect these antipatterns.

In Chapter 7 the evaluation will take place. First we list the research questions, after which we will zoom into each research question in a separate section. Related work is described in Chapter 8. Finally we conclude this thesis in Chapter 9. Here we list the contributions done and the implications of our work. We also list any limitations in our research and highlight suggestions for future work.

Additionally Appendix A highlights the flow of an e-commerce organization employing a microservice architecture. Apart from functioning as an example for the reader, this example is also used in the evaluation.

Chapter 2

Background

In order to understand the possible problems an organization might face when implementing microservices, it is necessary to have a general understanding on the background of microservices. This chapter aims to give the reader an understanding of this type of architecture.

First we give an overview on the design principles of microservices itself. Secondly we discuss how this architectural style influences the architecture of a system. Finally we conclude the chapter by describing how organizations themselves might need to change in order to maximize the effectiveness of the design principles and architecture.

2.1 Principles of microservices

By the lack of a current uniform definition of microservices in the academic literature, an overview will be given on the principles which underly the concept. These principles are taken from companies and software architects which have worked on microservice architectures. Since there is still a lack of established best-practices implementation details among the different architectures might vary. Further research is needed to evaluate the long term effectiveness of each part of the architecture in order to provide a solid set of best-practices.

The principles to describe microservices have been based on [James Lewis, 2014] and [Newman, 2015b]. These principles have been discussed with interviewed industry experts (as described in Chapter 3) which indicated these two concepts (modularization and resilience) were deemed most important in practice.

2.1.1 Modularization of microservices vs traditional approaches

In order to effectively understand why modularization is considered an important topic when dealing with microservices, we will first explain how modularization has been achieved over time. This historic context will show how microservices can be considered the *next step* in modularizing code.

Over time developers have learned how to build maintainable systems. Systems which have this quality often share some general design patterns:

- Loose coupling

- High cohesion
- Separation of data, logic, and interface

The maxim *loose coupling, high cohesion* [Hitz and Montazeri, 1995] is a good indicator of the modularization of code. When code is properly modularized, it offers useful interfaces to other components of the application, while the code within the module is logically strongly related. Over time there have been several ways to deal with this modularization:

- Creating a module,
- Creating a common library,
- Creating a service

No modularization At first if no modularization is present each part of the code is allowed to call each other part. Although this might allow for quick development initially there seems to be the tendency for all code to become interwoven. This practice is therefore not sustainable as this code gets exponentially harder to debug, refactor, and maintain [Abbes et al., 2011]. Every change in the code can have an unintended side effect in another part. This style is therefore not advised.

Creating a module When starting out with a piece of monolithic software, one generally encapsulates the module. This can be done using functionality of either a framework or the language itself (e.g. in Java the *package* feature). Using this approach components of the monolith can only access the external interface of the module. By reducing access to parts of the inner logic which are most likely subject to change, while exposing a stable interface, one can properly encapsulate the logic of the module [Booch, 2006]. By using modules the inner logic is still part of the program itself. Calls to the modularized code are done within the same process. Its code will also still be part of the build artifact. In case of a code change the entire application needs to be rebuild and deployed.

Creating a library In order to allow for easier reuse between different applications a module can be transformed into a separate library. Other code can depend on the external interface the library offers, while it remains agnostic about the inner logic. Libraries offer the possibility for sharing between projects and even organizations. Examples of libraries are client side api interfaces (to interact with a certain API service). Libraries are strictly outside of the applications code base, but are still part of the eventual artifact. Any calls done from the application to the program does not cross a process boundary. In case the library is updated the application needs to be rebuild and deployed. The library approach is very popular both by architects and developers, and entire ecosystems are in place to support this (e.g. Maven for Java, NPM for NodeJS, RubyGems for Ruby).

Creating a service The final step in decoupling an application from a component is by creating the component as a completely separate service. In this design style the code of the component is neither in code base of the application. Instead the service has its own build and deploy pipeline. Once the application requires a feature the service expose, it will communicate with this service (e.g. using a *Remote Procedure Call* (RPC)). The service processes the request, and optionally formulates a response for the caller. The inner logic of the component is now in a completely different process, and therefore inaccessible for the other application. If the interface of the service remains the same, the inner workings (including the platform, language, and framework) can be completely different from the original application.

2.1.2 Resilience

By splitting an application into several services, each of which runs in separate process, the likelihood of failure of one of these services increases. As noted in [Rotem-Gal-Oz, 2006] the network cannot be considered to be reliable. Even if the assumption is made that all the services itself are reliable, this means that services will sometimes be unavailable for other services.

There are three major reasons for services to fail:

- Programming error resulting in erroneous data (e.g. unparsable responses)
- The supplying service is offline
- The supplying service cannot be reached

The failure of a single service should however not impact the overall availability of the application. When each service with its network connections has an uptime of 99.5%, an architecture with 20 of such services will have an availability percentage of just $99.5\%^{20} = 90.4\%$. Services should therefore be able to cope with outages of other services by degrading gracefully whenever possible. Under some circumstances this is not possible and a service is required to deny the request (e.g. if a mobile broker cannot forward a request successfully to an API).

Teams building a service should therefore be aware how a failure of their service affects the end-user experience. Some services may fail and come back online without impacting the end-user experience at all. Other services may interact directly with end-users; failure of such a service may render part of the system unavailable. Apart from the above, a team also needs to aware of which other services their service will be dependent: which prerequisites need to be fulfilled in order to function at all? There are several ways for a service to handle failures:

- Declare itself either OK or failed
- Reduce functionality if upstream services are unavailable

The first option is often the easiest to implement and understand: a service can expose a custom *healthcheck endpoint* which can be used by other services to determine whether the service is indeed OK to respond to requests. Once this endpoint is called the service itself will ensure that its dependencies are available either by checking their

health (e.g. for services), or by checking the connection (e.g. for databases). In case a dependent service is unavailable, the service declares itself unavailable as well. This approach leads to the same problem as sketched above, since it effectively multiplies uptime percentages of all services. However it can be useful to remove unhealthy instances of a service from an instance pool (e.g. if the service has ran out of storage space).

The alternative is for a service to degrade functionality if an supplier service fails to respond, or does not respond in time. Degrading functionality can be done in various ways: one can fall back to cached data or one can omit data. The variant to use depends on the end user experience of the data. For example: if a warehouse service for an e-commerce system does not respond (timely), the website can handle this failure by either showing a cached stock count, or omit the stock indication altogether. In this case the availability of the system as a whole is not impacted when a user simply views the website, but he/she might be unable to actually order. An order microservice could deny the request for the specific item, since its availability is unknown. In this case the failure is contained to a specific part of the entire system (e.g. order fulfillment can continue to function).

Apart from preparing for failing services, some organizations actually force failures in their infrastructure to test whether their systems are capable of dealing with this. An example of this is Chaos Monkey by Netflix [Hoff, 2010]. A key component for managing failures is by monitoring services heavily. This point is further discussed in Section 5.4.4.

Additionally services should also prevent cascading failures: a slow downstream service should not cause another service layer to completely *lock* since all calls are waiting for responses. One can prevent this by using technologies such as *circuit breaking* and *bulkheads* within each service [Ranchal et al., 2015].

2.2 Microservice architectures

When hearing about microservices, many people see some similarities with the pattern which was popular in the 2000's: Service Oriented Architecture. The differences and similarities will therefore be described in this section. As a running example the architecture for an e-commerce site will be used which illustrates the architectural issues which may rise when using microservices.

2.2.1 Service boundaries

A key component for microservices is deciding which parts should be in separate services. A naive approach could be to promote each class to a separate service. This is effectively equal to taking the single responsible principle to the highest level: the application.

This level of splitting a problem domain does create some problems. Many services will depend on each other, creating a highly connected dependency graph. Therefore this solution is likely to require high maintenance. Another downside is performance related: calls between services (which do not run within the same process) are expensive compared to intraprocess calls. When a service needs to call multiple ser-

vices to perform its task, the response cannot be generated promptly and the network can become a bottleneck.

Instead a better practice seems to be to split the problem domain into *capabilities*. Capabilities revolve around sets of related functionality. This functionality is not confined to the technological structure, but to the actual corresponding entity in the business domain. An example can be a warehouse service: this service can be made responsible for managing inventory. It keeps track of all the products using a *Universal Product Code* [Savir and Laurer, 1975] and accompanying stocks levels (e.g. available and reserved). An e-commerce site can request the availability of a product to the service, and place a reservation once the customer completes the checkout. This reservation code can then be sent to the *Fullfilment* service in order to perform the actual shipment.

As one can note this warehouse service adheres to the practice sketched above:

- It keeps its own data (e.g. the stock levels) in its own database.
- An API with only the required functionality is exposed.
- An external service does not need to know anything about internal representations or logic.
- It is easy for other systems (e.g. Business Intelligence systems) to query the API without requiring major changes to other systems.

2.2.2 Communication

Communication between services can be done using several different methods. There is a distinction between synchronous and asynchronous communication. Synchronous communication is well known and understood by developers; a service sends a request to another service and receives a direct reply of this service (while the connection remains open). Asynchronous communication can also be used, during which the requester may receive a response at a later time once the callee has finished processing the request. Additionally there is a difference between *request-response* and *publish-subscribe*.

Synchronous communication is often used in conjunction with the request-response principle. A well known example of this is the current state of the web (HTTP). Asynchronous communication is also used a lot on the web using AJAX technologies (e.g. XMLHttpRequest). Asynchronous communication between services generally uses the publish-subscribe pattern. Services then raise events and publish these events to a central bus. Other services can listen to these events and react to it. Optionally these services can publish events on which the originating service can listen and continue processing [Etzion and Niblett, 2010].

Picking the right communication method is tricky and is likely to influence an architecture significantly. For some cases synchronous communication can have advantages, whereas other benefit mostly from asynchronous operations. For other scenarios a mix might be valid.

Synchronous communication (e.g. using REST) has a major benefit that it is well supported by tools and frameworks [Rodríguez-Domínguez et al., 2012]. Apart from

that developers are most likely to understand the paradigm, especially with traditional request-response schemes. Synchronous communication using publish-subscribe is possible, but does not provide additional benefits so it is almost never used. Asynchronous communication using request-response is also relatively well understood (e.g. communication with servers in Javascript is asynchronous) and support for it is on-par. Publish-subscribe can often be used in order to facilitate one-to-many messaging and to decouple systems. This pattern also suffers from less general understanding and it requires a central bus where messages can be published to (and be subscribed onto). The latter therefore requires more overhead in terms of additional services in the infrastructure. Examples of the latter are message buses such a Kafka or the more traditional ESB which was popular in the SOA era (ESBs will be discussed in Section 4.2).

Within microservices decoupling of services is an essential point. Apart from that many instances of one service can be running to enable load-balancing and facilitate fault-resilience. From the standpoint of decouplement, request-response does not seem ideal: services need to know explicitly which services to call. One change in a service can require an update of many other services. For requesting of information this is no problem since there should only be one authoritative service for each entity representation. However when dealing with status updates the service performing the update would have to call all services which might be interested in that action. The latter introduces a far larger amount of coupling compared to the first.

It can therefore be wise, depending on the number of services and the speed at which services can be added/modified, to make a distinction between two different types of communication. Therefore a distinction is made between two types of communication, each of which can be handled using another pattern.

The first type is a command: a service wishes to retrieve the representation of an entity, or needs to change some parameter about the entity. In this case there is only one service communicating with another. Apart from that the sending service is inherently interested in the response (e.g. the representation or whether the update was performed). For this use case request-response therefore is applicable.

The second case is an event of certain importance. When a change of a certain importance is made, other services can be interested in this event. The service which is authoritative for that event can therefore publish it to the central bus where other services can read and react upon it. The service raising the message does not need to know which services are interested in the message, nor how they deal with it.

An example of the combination of both for an e-commerce site is given for clarity. In this system there are four microservices:

- An order service
- A payment service
- A fulfillment service
- An email service

Once the payment service decides a payment was successful, it can communicate this with the order service using request-response. The order service updates its

representation and in turn sends a message to the bus indicating an *OrderCompleted* event. The email service can subscribe to this kind of messages and send an order confirmation to the user (after having retrieved the order using request-reponse). The fulfillment service finally also listens to the message, fetches the order data (using request-reponse) and prints a packaging slip. Note that the first request (between the payment service and the order service) could also have been performed by sending an *PaymentCompleted* event to which the order service reacts by checking whether that payment ensured the entire order was paid for. Depending on the exact architecture and requirements, both of these implementations are possible.

Also note that the order service does not even know about the existence of either the email service nor the fulfillment service. These systems are almost completely decoupled (except for the retrieval of information). An additional Business Intelligence system could also start measuring the timing of orders without any modification to the existing system (it can just subscribe to the bus). Due to the subscription pattern, the fulfillment service could also be taken offline for a day (e.g. warehouse maintenance) without impacting the remainder of the system. Once it comes back online it can start consuming the messages on the bus and catch up. This approach facilities a very fault resilient architecture with loose coupling between services.

2.2.3 Handling state

A core concept of microservices is the ability to easily deploy new versions of a service. Additionally one generally wants to be able to run multiple instances simultaneously (e.g. for load balancing or fault resilience). This requires that these instances share some part of their state or save it to another service.

The preference for many developers seem to consider stateful services (such as a database or queue) as *Backing services* [Marsden, 2015]. These are then managed outside the general application lifecycle. As the source suggests there seem to be some benefits to this approach:

- New instances can connect easily without having to synchronize a state first
- Existing instances do not need any reconfiguration for the new instances
- No longer required instances can be removed once no new requests are being processed

Keeping state out of the microservice can be imagined by the *pets vs cattle* concept [Tilkov, 2015b]. If an instance of application has no state within it and it goes down, one can bring it up again easily without any loss. This application can then be regarded as cattle since an operator does not care about the specific instance. However if an application with state goes down, data might have been lost or recovery of the specific instance has to be performed. This is considered to be a pet application since that instance had significant value for the operator. Managing cattle is therefore considered easier than managing pets [McKendrick, 2015].

The same goes with reasoning about services. When each request can be thought of independently it becomes easier for developers to trace the exact state the application is in at any given point in time. This can aid in debugging or reduce the time required to build the application.

2.3 Organizational implications

What does an organization have to change in order to fully profit of a microservice centric approach on IT? Microservices require a different approach compared to monolithic software. In order to take full advantage of a flexible architecture, teams should also be organized to seize this flexibility. Microservices are therefore often cited together with other developments such as cloud based hosting and a devops culture [Balalaie et al., 2015]. CI and CD are deemed indispensable [Newman, 2015b]. This section lists some key points which organizations need to be aware of before committing to the architecture. Failing to do so may result in a less effective architecture and development process compared to other methods.

2.3.1 Decentralization

As we will discuss in Section 4.3 about the use of a Service Oriented Architecture in practice, there seems to be anecdotal evidence that architects and developers should be given a larger degree of freedom when commissioning and decommissioning systems. The traditional view where an architecture is dictated and changes follow an internal change request has led to a situation where people have started to work around these procedures. According to Conway's Lay [Conway, 1968] a change has to take place from within the organization to effectively embrace microservices: decentralization.

When one opts for a three layer team (following the MVC pattern) where different teams handle the user interaction, the business logic, and the data persistence services will follow this model as well As explained in the previous section this is not a valid and scalable approach: services should resolve around business capabilities, not technical ones. A team should therefore be cross functional. The team should therefore actually *own* the service they wrote.

An additional point in decentralization is autonomy: teams should be allowed to create and deploy new services without long or complicated procedures. This prevents services getting *bulky* which was often done when using a Service Oriented Architecture which will be described in Section 4.3.

2.3.2 Capability vs helper

Some tasks are so common they can be extracted to separate services, such as email, scheduling, or generation of prints based on templates. These tasks can either be embedded in each service using a library or be separated to specialized services. Compared to regular *capability services* such services are called *helper services*: they execute specific tasks, which are shared across the entire organization.

By separating these concerns to specialized services the logic is kept in one place. For the caller the implementation details (such as extra dependencies or a specific programming language) are abstracted away. This makes the caller more flexible in its architecture.

The helper service can also handle more advanced tasks, such as email throttling or handling email bounces. This prevent the capability service to keep track of such state.

2.3.3 Automation

In order to take full advantage of an architecture where rapid deployment is possible, an organization should also facilitate behavior. Over the recent years there have been multiple developments which can facilitate this: a devops culture [Httermann, 2012], continuous integration [Duvall et al., 2007], containerization [Turnbull, 2014], and cloud computing [Katzan Jr, 2009] are examples of this.

A devops culture embraces the concept of *ownership* of a specific piece of software. This approach advocates that a team develops a service, maintains it, ensures it works in production (and handles service failures). An opposite customary is a situation where a certain team builds a service, which is then transferred to a maintenance team while an operations team is responsible for running and scaling the application. The first situation allows for a culture where the development, test process, and releasing of new versions can happen in a more frequent and reliable matter.

The devops culture embraces the automation of software delivery and deployment. Additionally it focuses on a culture where most things should be programmed (including IT operations). Tooling such as Docker (for containerization), Jenkins (for continuous integration), Vagrant (for development environments), and Puppet (for scripted server provisioning) help with this.

Finally by embracing virtual servers over physical ones one can simplify IT operations in production significantly. When a new version of software is deployed (or a configuration change to a server has to be done), this change is applied over each server one-by-one. This leads to a phase where not all servers are consistent with each other, and bugs may occur. By using virtual servers (which can, but not have to, be rented from cloud companies) one can embrace the concept of an *immutable server*. This is a server which is never modified, nor for new software versions, nor for configuration changes. In case of such a change, a new server is brought online and the old one is stopped. For a brief period there may be two clusters running but these cluster are completely homogenous in software and configuration. This may reduce the change of failures due to these changes. Additionally the practice can be made completely automated. With purely physical machines such an architecture would be much more difficult and costs would grow quickly (due to the excessive number of servers standing by).

Chapter 3

Interview participants

As an important part of this thesis a number of interviews was conducted with key technological persons of different companies. This was done in order to obtain first hand knowledge on the use of software architecture within the industry itself. These interviews form a common thread during this thesis.

Motivation In theory the downsides of bad maintainability, technical debt, and antipatterns are well understood. However from industry experience, it may seem like there is a discrepancy between these described downsides and the actual experienced downsides.

In order to get a clear and first-hand indication on how architectural design, architectural defects, and antipatterns influence organizations and their business, these organizations had to be asked about their experiences in this regard. The main point of interest was how the chosen architectural design influenced the way the organization as a whole worked.

Interview setup As shown in Table 3.1 a total of 15 companies were selected which came from the following industries: finance, retail, energy, staffing, IT, logistics, trading, and telecom. Each of these companies explained their use of software architecture in their organization.

Based on earlier research [DiCicco-Bloom and Crabtree, 2006] we decided to conduct the interviews in an non-formal style: interviewees would be regarded as participants to the conversation instead of just being asked questions. In order to let the interviewees be honest and open about the IT challenges of their employers, the choice was made to conduct these interviews anonymously. This was also done to ensure participants were able to talk openly about competition sensitive information. Therefore the names of the participants nor the organization names will be published. With each of the participants a talk was held ranging from 30 minutes to 2 hours.

The results of these interviews will also be used in Chapter 4 and Chapter 5, depending on the organizations dominant style as shown in Table 3.1. With a number of relevant companies, a design evaluation of anti-pattern regarding message buses and system design was held at a later stage. The results of this will be used in Chapter 7.

It should be noted the author of the thesis has strong relations with organization *C3*. Interviews and implementations of systems with this organization have not been performed by the author himself.

Table 3.1: Participating organizations

Company	Sector	Dominant architectural style
C1	Finance	3 systems, file based communication, relatively monolithic (Section 4.1)
C2	Finance	40 SOA [Perrey and Lycett, 2003] services (Section 4.2), event bus communication (Section 2.2.2), single code base
C3	Finance	15 services, 1 monolith, Kafka/REST (Section 6.1 and [Fielding, 2000]) communication
C4	Staffing	10 services, 1 monolith, Kafka/REST communication
C5	Finance	Monolithic applications with ESB (Section 4.2), outsourced IT
C6	Energy	Mostly monolithic, moving towards microservices (Section 2). REST based communication
C7	Retail	Microservice based architecture, restful communication
C8	Retail	Monolithic software, supported by small services
C9	Telecom	Core systems are monolithic, remainder is created by separate business units. Communication is done using file based batch exports
C10	Trading	Specialized monolithic systems
C11	Logistics	Lots of small systems, with a core IT team maintaining them. Communication over ESB systems
C12	Retail	IT based on AWS [Varia, 2010] with immutable services and containers, except the monolithic main site itself. REST based communication
C13	Retail	Multiple monoliths which works quite independently
C14	IT	Makes multiple systems for customers, nowadays mainly focused on microservices and event bus communication
C15	Finance	Several old monoliths, but with more recent smaller microservices on top

Chapter 4

Monolithic and Service Oriented Architectures

In Chapter 2 an explanation was given on microservices themselves. From the interviews, conducted as described in Chapter 3, it turned out that organizations employed two other architectural styles: monolithic and Service Oriented Architecture. These strongly differ on several viewpoint, e.g. their development, concurrency, operational, and deployment viewpoint [Rozanski and Woods, 2011].

The goal of this chapter is to compare the architectural style of monolithic software with a Service oriented architecture. In order to understand how microservices are used in practice, we also need to understand how the most dominant alternatives are used within the industry. For both monolithic software as well as service oriented architecture a section will explain the key focus points, industry usage, the advantages, and the disadvantages.

For this comparison the talks with the organization experts of Chapter 3 served as a basis. During each of these interviews the present member(s) of the organization was/were asked a fixed number of questions. Based on the answers on these questions we asked more in-depth follow-up questions. After these were done the architecture was discussed in more detail. The interview concluded with a discussion about issues or problems that were faced with the architecture, and a reflection on earlier design choices.

In order to structure some of the interview questions, each of the participants were asked the same set of questions (listed below). Additionally we asked follow-up questions based on the previous responses. The starting questions for the interview were:

- Can you describe the architectural style used?
- Why was this architectural style chosen over other styles?
- How does the chosen style impact the product development?
- Is IT considered a crucial or supporting part of your organization?
- What are the development plans for the foreseeable future?

4.1 Monolithic architectures

The term *monolithic architectures* was often used by the industry experts to describe their software architecture. A more in-depth analysis of this architecture is therefore just.

Monolithic software is software where a single program performs all the tasks as set by its requirements [Stephens, 2015] . It therefore does not have dependencies on other applications to fulfill its tasks. Examples of these tasks can be showing a user interface, or persisting data to a storage device. Monolithic applications are therefore regarded to be *self contained applications*.

While the application does not depend on other external applications, it may still use other modules. These modules however are internal to the monolith itself. An example is a library for persisting data to a database; this library is internal to the application, but not part of the application code base.

The term monolith is becoming more difficult to use consistently, since it is also used for different kinds of software which do not fit the exact definition. One of the reason for this is the *three-tier model* [Smith et al., 1998]. This model has become more dominant with the rise of *Web Services*. In this case there can be an application which handles the business rules (e.g. an *API*), a client which handles the user interface, and a storage service which persists the data (e.g. a *DBMS*). This architecture does not conform to the classic definition of a monolith. Instead one often refers to the API as a monolith if it consists of a single application which handles all the business logic.

In the remainder of this thesis the term monolith will be defined as *a self contained application which meets the complete set of design requirements by itself*. A service oriented architecture employs applications which work heavily together to fulfill their set of requirements. This definition includes three-tiered software. Note there is no restriction on the code organization, which allows concepts as a *monorepo* [Durham Goode, 2014], where all applications share a single code base, but different artifacts can be created from this code base.

Examples of monolithic software are early versions of Microsoft Word and Mozilla Firefox. However more recent version of Word share some functionalities with other Microsoft Office products.

4.1.1 Industry usage

During the interview phase as described in Chapter 3, each of the 15 companies indicated that they had at least one monolithic system supporting some needs of the business. This is a clear indication that monolithic software is not uncommon within the aforementioned industries. Of all the monoliths which still were in place, several were originally developed in the 1980s, whereas the development of the most recent monolith commenced around 2013. The largest number of still active monoliths has been developed between 1995 and 2008. The average monoliths lifetime is between 6 and 8 years (however the standard deviation of 5 years is quite high). This number is shifted slightly due to a number of systems originating from the finance industry which have lived considerably longer than the average. The average lifetime is likely to be strongly dependent on the exact industry. The main reason for the decommissioning

Table 4.1: Age of monolithic software

Company	# monoliths	Average age	Active development?
C1	3	5 years	Yes
C2	1	9 years	Yes
C3	1	2 years	Yes
C4	2	3 years	Yes
C5	11	13 years	Yes
C6	4	11 years	No
C7	2	4 years	Yes
C8	4	5 years	Yes
C9	-	-	Yes
C10	3	2 years	Yes
C11	14	7 years	Yes
C12	1	2 years	No
C13	3	9 years	Yes
C14	-	-	*n/a*
C15	28	18 years	No

of these systems is either the ever increasing maintenance costs, or the fact that the system is getting too far behind on the technology curve. Examples can be supporting responsive web layouts or mobile apps in addition to a web page. For companies which consider IT of major importance in their services, software tends to live shorter before being replaced.

The retail industry (most interviewed companies from this industry fall into the *e-commerce* group) considers software development crucial to their business. Within the finance industry the software development strategy strongly depends on the age of the organization.

4.1.2 Advantages

Each of the companies has also responded on what they consider to be advantages of monolithic systems. For some of them, these advantages are applicable, whereas for others some advantages are less applicable.

During the initial design of a system many module boundaries are not entirely clear within the problem domain. Using a monolithic approach allows for relatively easy changes; moving or changing boundaries can be done without revising the internal communication within the application. As noted by C1, C3, C8, and C14 this advantage can significantly reduce the initial design phase by up to 30%. These participants also pointed out that in a fast moving industry, being faster to create systems than a competitor can be very important to remain competitive. Some organizations therefore (sometimes) opt for monolithic systems just to allow very fast design phases.

Another mentioned key advantage is the experienced ease of development of such a system (by C1, C3, C4, C8, and C14). Many frameworks, libraries, and tools work best when working with the full systems code, and are therefore well suited for a monolith. Many frameworks come with support for a variety of tasks out of the box, such as database access, ensuring locks between several instances, and have an entire *Hierarchical Model-View-Controller* [Cai et al., 2000] embedded. This allows devel-

opers to quickly start developing the software, without losing any time building the building blocks of the application. The same arguments applies to tooling; many IDEs have support for maintaining code quality within a single code base, and allow easy refactoring within that same code base. The discovery of unused endpoints and unused methods can therefore become a task of the IDE instead of the developer. Using automatic refactoring tools within an IDE such as *IntelliJ* therefore allows for quicker iterations once code needs to be added, split, or removed. Another result of this advantage is the availability of developers for a system: many developers are used to the practice and can therefore start to contribute quickly to the product after the hire. For fast moving startups, this can be an advantage since new developers have a gentle learning curve. The same goes for other industries for which the larger number of possible developers is an advantage.

The third main argument in favor of monolithic software is the reduction of communication with external systems which was mentioned by C3, C4, and C8. Most communication is maintained within the program, and therefore regular method calls can be used (external communication can be calls to partner software). Many steps such as data serialization between components are not required, making it easier to reason about the state a program is in. For some industries, such as trading, an additional benefit is that internal communication is much faster when compared to network based communication (for high frequency trading, milliseconds really matter). One also does not need think about the resilience of the network for the application itself (the usage of the application might require this to some extent though).

The final argument is the ease of setting up such an application in production, including any deployments after the initial version. For C13, this was a concern, since it cuts down on operating costs. One can simply stop the applications, fetch the latest version, and start it again. This form of deployment is straightforward and therefore easy to implement and less error-prone.

4.1.3 Disadvantages

However the interviewees indicated there are also some sections where monoliths do not perform as well. Depending on the industry these can be negligible or very important.

The first disadvantage according to the interviewees of C2, C3, C4 and C7 is that components in monolithic software have the tendency to become increasingly coupled other components of the same monolith. At some point in time, this might hinder the further development of the application. If this technical debt [Buschmann, 2011] is not paid in a timely matter, the maintenance costs of the application are likely to keep on rising. Apart from the costs for maintenance, the implementation of new features can also become increasingly difficult over time. This seems in contrast with the earlier stated advantage of cutting the design time: however the implementation is regarded as development time, not design time. A win in one of these phases may therefore eventually lead to a loss in a later phase.

Participants C2, C3, C4, C7 and C12 also mentioned the practices of scaling and fault-resilience. These can both negatively impacted by a monolithic application. Due to the setup of the system, it is hard to scale independent features of the application without also scaling other parts. Since scaling the non-useful parts as well will

consume more resources, this yields higher costs for the infrastructure on which the application depends. Since the application is big as well (compared to feature which needs scaling) the scaling process can take significantly longer, making it difficult to scale the application elastically with the demand. Scaling of a single machine has its limits (the computing power and memory in a machine is still limited), and proves to be a single point of failure. The same goes for fault-resilience; once a common component fails the entire application is brought to a halt (or may even crash), instead of degrading functionality.

The final disadvantage (as stated by C3 and C4) is the deployment of a monolithic application (although it was also mentioned as an advantage). At some point in time (as the application grows) the deployments take increasingly longer. Similarly deployments have to be done for each change, no matter how small the change set actually is (for example in Java adding a log statement changes the resulting JAR file). Due to possible downtime while deploying, some organizations tend to deploy on a fixed-time basis (e.g. once a month). This brings another downside: in case of a failure in the newly deployed artifact one cannot pinpoint an exact change set. In such case a rollback may have to be performed, which will delay new features and fixes.

4.2 Service Oriented Architectures

In a Service Oriented Architecture (SOA) smaller applications are employed where each handles a subset of the set of requirements. Instead of applications the term *services* is most often employed, since individual services are not capable of working stand alone. These applications communicate with each other to perform a common task. This can be done using direct service communication (e.g. using *REST* or *SOAP*), or over a central bus system (e.g. an *Enterprise Service Bus* or *Kafka*).

The service oriented architecture is based on a number of key principles [Krafzig et al., 2005]:

- Loose coupling: Services should minimize the number of dependencies between each other,
- Abstraction: Services may only use each others public interfaces but should not be aware of any internal logic,
- Reusability: In case of common components a separate service should handle the common logic in order to advocate reusability.
- Statelessness: Services should not maintain more internal state than strictly necessary (state should be written to a database or messages should be sent to other services).

SOA was promoted strongly in the 2000's by companies as IBM and Gartner, which offered many services and products to facilitate working with this architecture (such as consultancy or Enterprise Service Bus products). Over time, SOA became synonymous with protocols such as the set of *Web Services* and *Extendible Markup Language* (XML). The Web Services pattern offered protocols for each of the concepts

of SOA; examples are service discovery, message passing, authentication, authorization, and encryption (among others) [Alonso et al., 2004].

Any data should be localized to one service, which is responsible for managing that data (this pattern is known as *Information hiding* [Parnas et al., 1983]). Any other service wishing to access the data, should call the service which holds the data to obtain a representation of that data.

In short SOA takes the five *SOLID principles* [Martin, 2003]. These are then applied to the architecture itself:

- Single responsibility principle: each service should have a single responsibility and call others to obtain information,
- Open/closed principle: a proxy service can be placed in front of the actual service to enrich the data of the service itself without modifying existing clients,
- Liskov substitution principle: a proxy service can be placed in front of another service to handle more specific calls to that service,
- Interface segregation principle: each of the services has a specific client interface instead of a larger common one,
- Dependency inversion principle: services should only depend on each other interfaces, and not on the internal logic of other services.

4.2.1 Industry usage

A smaller set of the interviewed organizations (9) has indicated that in addition to several monolithic systems they also employ a service oriented architecture. This architectural pattern is therefore less likely to be used when compared to monolithic software, although this may depend on the industry the organization is in. There is a bias here since a large percentage (8) of the interviewed organizations considers IT as crucial. The likeliness of the usage of SOA is therefore probably lower when the entire software engineering industry is considered.

Services in a SOA-based architecture have very different lifetimes. Some simple services have a lifetime of over ten years, whereas others are replaced within months after their entry into service. The explanation for this is that services at some point can be considered to be *done* (so no more changes are made and the service keeps running), or *expired* (after which the service is most often replaced) which was indicated by C2. The lifetime metric therefore does not convey any useful information.

There seem to be two predominant ways for services to communicate in a Service Oriented Architecture: *synchronous communication* and *asynchronous communication*. These communication styles can be further divided into *request-reply* and *publish-subscribe* [Eugster et al., 2003] patterns. Request-reply is most often associated with synchronous communication. A well known example of this is browsing the web (a web browser sends a request and receives a web page as a result). Publish-subscribe is often used in an asynchronous fashion. In this scenario a service places a message on a bus, which is sent to any service which has subscribed to this type of message. A service may respond by sending a message back over the bus system to the caller.

From the point of loose coupling a bus is considered favorable by C2 since it prevents services to depend explicitly on each other. This also allows a service to go offline for a while, and to do a catch-up of messages once it comes back online (or allow load balancing features between different instances of the same service). Although this pattern does decouple the services from each other, it couples each system to the bus itself. Within the set of interviewed organizations, bus based systems are predominant (hence using asynchronous communication following the publish-subscribe model).

4.2.2 Advantages

There are several reasons to pick a service oriented architecture. The extent to which these advantages apply to each organization or industry can differ significantly.

Thanks to the application of the single responsibility principle, a service is easy to understand for a developer. Each service handles a set of strongly related tasks, and defers tasks outside its scope to other services. The inner logic of the service is therefore limited. Especially if services are well designed and conform to the the dependency inversion principle as well, services can also be swapped with other implementations of the service as long as these conform to the same external contract. This advantage is considered important by C11 and C14.

A second advantage is that due to the decoupling of services, it is easier to prevent a lock-in to a certain technology stack (e.g. a certain DBMS or a programming language). Depending on the exact needs of a service, one can easily pick the *right tool for the job*. This allows for better performance for individual services as stated by C11. Although this advantage is often quoted, all applicable companies have indicated that they limit the technology stacks to some extent. This is done to prevent the occurrence of stacks for which the number of developers is severely limited.

Thirdly certain parts of the application (e.g. specific services) can be scaled individually depending on the demand for the specific service. These scaling purposes have been important in the past for C2 and C11. By only scaling certain parts one can be specific which services should have a higher instance count. This can be done based on the load on such a service or for reasons as fault-tolerance.

Another mentioned advantage is that deployment of the individual services is easier to realize, since one does not need a so called *big bang deployment* but instead can deploy the specific service containing a change. At C2 this reduces the time required to bring a service from development to production. In case of problems with a new or changed service the service can be rolled back to an earlier version faster compared to a monolithic application.

Finally due to the nature of the small services, with highly specific interfaces, testing is easier to perform (which was of concern to C11). All services are tied to specific interface for their calls. These interfaces can be independently tested to check whether they conform to the specifications. Assuming other services work as expected, any failures to comply are easier to pinpoint due to the smaller amount of inner logic.

4.2.3 Disadvantages

Although the advantages all seem legitimate, there are some issues with a purely service oriented architecture.

First of all it is increasingly expensive to get the service boundaries right. In case one later decides to split or merge services, this can have significant impact. In case service boundaries are off, this can result either in bad containment of functionality or overly chatty communication between several services [Erl, 2008]. This has been experienced by C11 and C14 when performing initial designs.

Secondly developers at C2 still feel there is a significant lack of proper tooling In case of a problem or bug within a monolith, a developer can start a simple debug session in the IDE and step through the code. For services one might need multiple debuggers to step through the set of services which are used. Especially timing related issues or race conditions are harder to debug. Some interviewed companies such as C10 have therefore kept certain time critical systems monolithic, simply to prevent variable timings of communication.

A major disadvantage which was experienced by C3 and C4 is SOA is that communication is now a key component. Due to the decoupling of services, the communication is no longer contained within the process. This means there is an additional time and performance penalty for communication. As mentioned before, if the service boundary is not well defined, this can result in overly chatty communications. Since each of these communication calls is more expensive than the equivalent intraprocess communication, this can result in performance problems. Another possible problem with this form of communication that a service can no longer guarantee another service is running. Therefore each service needs to be able to detect unhealthy states of the services it depends on, and act accordingly to its caller. The communication can also be more difficult to reason about (e.g. determining a global state of a distributed network [Chandy and Lamport, 1985]).

Testing individual services has become easier, but testing the system is increasingly harder as stated by C2 and C7. One can imagine a situation where a test of system *A* is done with respect to system *Bv1*. However once this test succeeds, this is no confirmation that system A will work well with the just released system *Bv2*. For organizations which tend to deploy new versions of their services often, this might mean that a CI tool is always be behind the versions running in production. Another downside can be that for a complete test of service *A*, all downstream services of A should either be running or be mocked. Running all these services would mean that a lot more infrastructure is required for testing, while still not deviating the problem of newer versions of the service running in production. Mocking has an additional problem: apart from not being necessarily up to date with the actual system, the mocked code is part of system *A* itself. In case of an update of service *B* on which *A* depends, the mock code for *B* needs to be updated in all sources of dependees of *B*!

4.3 Retrospective on Service Oriented Architectures

Many organizations have opted for the event bus (usually an Enterprise Service Bus [Chappell, 2004]) combined with publish-subscribe patterns. This Enterprise Service Bus has several features, such as mediation, routing, transformation, and security. At first many organizations such as C5, C6 and C11 considered this as an advantage for the ESB since it allowed them to keep this logic in a central place. The definition of SOA suggests that for each set of coherent features or tasks a separate service should be

developed. According to Conway's Law [Conway, 1968] organizations that are asked to design systems will eventually design systems which are effectively copies of the structure in the organization itself. In order to apply a Service Oriented Architecture, this means that for developers it should have been easy to add or modify parameters of the central bus. However in most organizations this system would be placed under the responsibility of the central IT division, which did not develop the remainder of the software. Due to the lockin with a centrally managed ESB, it became difficult to easily add new services to the system. Therefore developers and architects alike started to opt to extend existing services to handle the new logic. In the end for many organizations ended with several large monolithic applications which communicated over a central bus system (for example at C5). This meant many of the advantages of Service Oriented Architecture were lost, and replaced by the disadvantages of monolithic software. Eventually this resulted in large and inflexible software, of which maintenance costs started to rise rapidly after a certain amount of time.

Looking back to the paradigms of SOA, many still consider the architectural patterns valid. However due to all the constraints in SOA (e.g. ESBs and the complex Web Services standards) these principles could not be applied effectively in organizations. It therefore seems the initial start of SOA did not deliver its promises.

Chapter 5

Microservices in practice

As a part of the conducted interviews as described in Chapter 4, relevant organizations have also been asked about their experiences with microservices. These organizations have indicated how they use microservice based technology and how it influenced their general development process. This chapter therefore gives a deeper understanding of the advantages and disadvantages of the architecture. It also gives a first answer how organizations employ the communication between services in their systems.

Obtaining knowledge first-hand of experienced organization which use microservices gives clear insight in the decision processes, experienced advantages, and disadvantages. Since this topic does not (yet) have a fair amount of published literature, this chapter will therefore give insight in this, untill now, relatively unknown topic.

6 of the 15 interviewed organizations adopt some form of a microservice architecture within their application infrastructure. It should be noted that this does not require that the entire technology stack is developed around microservicing. Some pieces of monolithic software may still be around, although most have adapted these systems to work within the new landscape. Only organization *C7* has completely switched to a microservice architecture.

When discussing the different implications of picking this architecture, the following points will be touched:

- The reasons to pick the pattern,
- What the architecture looks like now,
- The resulting influence on development,
- The advantages which have been experienced,
- The disadvantages which have been experienced.

5.1 Decision process

In order to understand the following sections one first should understand why these organizations have opted for a microservice approach for their software architecture. As explained in Chapter 2 and Chapter 4 there can be multiple reasons to pick a certain architecture. The relative importance of these reasons also depends on the organization and the industry the organization is in.

None of the organizations which are using microservices started out using this approach. The primary reason for this was that the organizations already existed before the paradigm had started to become popular. Due to issues with the Service Oriented Architecture (which had been in place for one of these companies), these organizations started out using monolithic software development approaches.

One of the key components of microservices is that it strongly separates responsibilities. The resulting code should therefore be easier to maintain, potentially leading to a faster delivery process for new services and features for existing services. For organizations *C3*, *C4*, *C7*, and *C12* this was the fundamental reason to adopt a microservice architecture. The development speed of their monolithic architecture was decreasing due to the increasingly more difficult process of overseeing the effects of a change set. This can either cause bugs to manifest themselves in a production environment, or lead to a slower delivery cycle. For these organizations the ability to quickly iterate was defined to be key, therefore the architecture should allow for short development cycles with frequent and automated releases.

An additional component for a few of these organizations (*C3*, *C7*, and *C12*) was to allow flexible scaling of services. Each of these organizations fall either into *Finance* or *Retail* industries. The online activities of these companies have a highly flexible demand; the ability to scale a service within minutes is therefore deemed crucial in order to balance availability, performance, and cost. A company as *C12* can see a sudden surge in traffic of over a factor 50 within several hours, whereas *C3* indicated to sometimes see a (planned) demand change of over a factor 100. In order to prevent lots of idling servers, these organizations use server instances which can be started quickly and scale flexible with the demand for the service.

Finally a reason for adopting microservices is the ability to quickly test a smaller service compared to a larger one as stated by *C7*. This reason strongly related to a faster delivery process, since the feedback loop between a CI system and developer is greatly reduced [Duvall et al., 2007]. As described by some organizations, a quick test feedback loop can also result in a higher effectiveness of less-qualified programmers: one does not need to oversee each change, but can rely on testing for this.

The majority of participants did not use code quality metrics (e.g. low coupling or test coverage) as a goal on itself. These metrics correlate to the state of the software and function as indicators of shipping speed and errorless software, but are not conceived as business goals. Other metrics, such as reliability, uptime, or time-to-market were mostly used to get insight in the state of the software.

5.2 Resulting architecture

Since each of the aforementioned companies already had working software in place, the architects and their teams were already familiar with the specific domain. As described in [Tilkov, 2015a] it can be very hard to create these boundaries right (and the cost of changing the boundaries is high when using microservices, see Section 2.2.1). The best situation is therefore described as *the ideal scenario is one where you're building a second version of an existing system*. These organizations did just that: take a look at the problem domain from existing software and model the boundaries between the different subsystems. After the modeling of the business processes, one can

start with development of the separate services.

Interestingly enough each of the different companies has opted for a similar approach. These approaches share the following principles:

- All data is JSON encoded,
- APIs are setup using representational state transfer (REST) [Fielding, 2000],
- Services are split across business processes, not technical entities,
- Services emit events based on certain actions,
- Services share an authentication and authorization layer,
- Services are automatically built and deployed,
- Services are small (a service is defined as small if *it can be rebuilt by the team owning it within a period of 3 weeks*).

However there are also a number of differences. Within *C15* all end-user services communicate with the different services directly. These services may then continue to communicate with the respective backends. This process is known as *Backends for Frontends* [Newman, 2015a]. Other organizations let all requests go through a general API which directs the requests to the individual services. In the latter case only a single API is exposed, compared to multiple different APIs.

Another different is how handling state is done. Some organizations eliminate every internal queue, by deferring tasks using a message bus (and consuming them in the same service). This allows for flexibility within the service, since a task can be handled by another instance than the original one. It therefore functions as an internal load balancing system. It is also more fault resilient since an error in a service does not cause any loss of state. Other organizations tend to keep some state internally within queues to handle those at different threads. When decommissioning an instance this instance needs to either finish its work, or push the queues back to a shared medium (e.g. a database). The latter system is easier to implement and is closer to the development style used in monoliths, whereas the other is more fault resilient and scalable.

In conclusion the resulting architectures for microservice approaches vary between the different organizations. What can be right for one organization can be overly costly or complex for another one. In its essence this is a similar situation to monolithic software approaches.

5.3 Experiences

This section will explore the actual experiences of the organization when using microservices. It will take a look at how well the practice aligned with the reasons to pick the pattern. Additionally it will look into other experiences advantages which were not expected. Finally it will take a look at the (non-expected) disadvantages when used in production.

5.3.1 Advantages

As stated before, for organizations *C3*, *C4*, *C7*, and *C12* the primary reason to adopt this approach was the expected decrease in the time required to go from development to production. Each of the interviewed organizations has experienced this benefit (it should be noted that all of these organizations deliver services through the browser). Due to an improved delivery pipeline, it also became possible to automatically launch a deploy process for very minor impact issues (e.g. a spelling error in a localization file). In previous processes this was not possible, and many of the smaller defects or features had to be released as a part of a release train [Leffingwell, 2007].

Although not a prime reason to switch architectures, the number of defects or faults in the software also decreased for *C2*, *C3*, and *C4*. This was partly attributed to more rigorous monitoring (as will be explained in Section 5.4.4) but also due to the adoption of a *devops* culture as well. Such a culture make developers and operators work as part of a single team, so errors in production are fixed faster (and the actual fix is released faster). Developers are also more aware how the applications should scale.

The increased simplicity of the each of the applications also has measurable impact on the effectiveness of (new) developers which was recognized by *C7*. They are able to quickly start working on parts of the application set, without having to know anything about the inner workings of the software they interact with. This allows employees to roam more freely over the different projects within an organization while they can maintain their productivity. The organization itself is therefore able to react quicker to changes in the business.

Another key advantage is of a non-technical nature: developers feel much more involved in the organization as a whole which was mentioned by each of the organizations employing microservices. Their team is empowered to make decisions on its own without a need for Change Reviews by upper management. Instead an idea can be built, tested, and evaluated within days instead of months. The lack of a formal Change Review process also means an organization is able to cut down on bureaucratic costs.

5.3.2 Disadvantages

Unfortunately there are also some disadvantages for this architectural pattern. Some of these were already clear before organizations embarked on the road, some emerged later on.

Whereas the complexity of microservices themselves have diminished, there is an additional penalty involved: the operations overhead. This turned out to be significant for *C2* and *C4*. When dealing with a single artifact one can relatively easily deploy and run the application. Once dealing with microservices, one also might need additional scaffolding for message passing, load balancing, and orchestration. Compared to a monolith which handles the same business logic, this is a significant up-front investment. However this investment is mostly considered unavoidable as deploying new versions of dozens of services manually each day is also deemed ineffective. *C2*, *C3*, and *C4* did indicate this up front investment does pay off over time, but the initial time and cost required for the pattern is something that should not be taken lightly.

The previous points also reflects on the teams themselves: developers need a better understanding of how operations works (*C4*). This reduces the pool size when

hiring new developers, or requires more training for these developers. Each of the interviewed companies has indicated that finding the *right* developers is hard. Adding more requirements for job positions therefore makes it even more difficult to fulfill vacancies.

One of the main advantages of microservices is how easily they can be refactored internally, without touching services that depend on it. However as explained in Chapter 2 and stated by *C7* this is not the case when changing service boundaries or contracts. In both cases one needs to verify multiple services and adjust these to the new boundaries or changed contracts. These changes across services are highly coupled, so a big release of multiple services within a small timeframe is required for this. Alternatively one can support multiple versions of services and migrate dependent services more slowly. The first is difficult from an operations point of view, whereas the latter is technically harder.

Another downside is the so-called *communication penalty*. This is also present in other forms of distributed software development and covers a wide variety of possible problems for such architectures:

- There is no transaction safety present,
- Latencies between components can vary more compared to interprocess communication,
- Additional complexity is involved in message passing or serialization of data,
- Asynchronicity is difficult to reason about,
- Debugging a distributed application is hard [Manabe and Imase, 1992]

Developers need to be aware of the pitfalls of a distributed architecture, and need to deal with these adequately to ensure the system is up, performant, and available. However most developers (*C7* and *C14*) are not familiar which such architectures, and therefore need to gain some training or experience with this first.

5.4 Influence of the architecture on development

A software architecture has a certain impact on the development process used within an organization. As described in Section 5.1 the acceleration of the development process was a key reason to move to a microservice architecture. This section will explore what the actual impact of the the architecture has been on several parts of the development process.

5.4.1 Development speed

The term *development speed* refers from the time required from building a system of feature, to testing it, and bringing it live in production. As stated in Section 5.1 this was a major concerns for organization to switch to microservices.

There are several ways in which development speed can be affected:

- Bugs can be fixed faster,

- Features can be tested with a subset of customers,
- New services can be deployed to production faster.

As noted by interviewees of *C2*, *C3*, and *C15* the time required from building to shipping has effectively reduced this time. Where some organizations used to be able to ship code at most a few times a month, these are now able to ship several times a day. This allows for faster iteration of features, and more effective *A/B testing* [Siroker and Koomen, 2013] of applications. The possibility of increasing clicks or revenue provides direct quantifiable results for the business. As a result of this faster development speed also strongly relates to the business goals.

5.4.2 Technology stacks

Although microservices technically allow to use any technology stack possible (*pick the right tool for the right job*), each interviewed organization limits the number of different technologies. Instead of this *polyglotism*, most organizations seem to focus on *consolidation* (such as *C2*, *C3*, and *C4*).

Consolidation offers a number of advantages for both organizations as well as developers:

- Since the cost of learning new technology is high, staying with known technology avoids this cost,
- A team can know enough about a few technologies to write and operate their software effectively, but cannot do this for a large number of technologies,
- Developers within an organization are more flexible if their knowledge is transferable,
- Operations can be done cheaper on a smaller number of well known technologies.

However the actual technologies can still vary between the different companies. Some strongly rely on Java, whereas others run anything as long as it is based on the Java Virtual Machine. The same goes for database technology: some organizations run on Oracle, PostgreSQL, or MySQL for relational data. For document storage other technologies such as MongoDb or Elasticsearch can be used. However an organization generally picks one single database technology within a specific domain.

Hence one can conclude that the languages, frameworks, and data services are often constrained. This does not go for all internal facets of a service. With most companies developers are free to pick any dependency (and version of a dependency) as they prefer. Since the amount of code is relatively limited within a microservice (especially when compared to a monolith) the likelihood of dependency conflicts is also lower.

In conclusion one can say that the advantage of freedom is often constrained by organizations. There is a strong consolidation to a set of technologies whereas within a service developers are free to use other parts.

5.4.3 Automation

In order to achieve the goal of faster development speed (as described in Section 5.4.1) a certain amount of automation is required. Most organizations (*C2*, *C3*, and *C4*) indicate they already had some internal automation in place, but each organization with microservices has indicated this automation was extended to use them effectively.

The following processes have been the key candidates for automation:

- Unit testing,
- Continuous integration,
- Artifact building,
- Deploying.

The first two of these activities are usually already automated using available tooling such as Jenkins and JUnit. Due to the slower release cycle, the latter two activities were usually not a prime candidate for automation. Often the build process had changed slightly between big releases, making the reuse of existing scripts difficult. In order to take advantage of the faster development speed, these processes now also need to run automated (since otherwise there is little gain).

Depending on the language, platform, and framework the building of artifacts may be easy or complicated. Some applications can be distributed as a single *Java ARchive* (JAR) file, whereas other applications might need to be packages as a *Virtual Machine* or as a *Docker image*. Software architects need to keep this in mind while developing the application (e.g. with regard to packaging dependencies or external services). While writing a service it becomes a key concern for a developer how the service itself will be built and deployed. The build steps are therefore mostly written as part of the application itself.

Handling the automated deployment of a service is also deemed useful by all interviewees. This allows for new stable builds to automatically go into production without developer or operator interaction. For the deployment one needs to consider several different steps: how to handle database changes, how to perform rollbacks, or how to perform a smoke test [McConnell, 1996] before live traffic hits the new version. For each of these there are different possibilities (e.g. [De Jong, 2015]) depending on the exact needs of the application. The behavior of the service during such steps needs to be defined so other services know what to expect. Automated deploys can therefore vary from a *git checkout* or by bringing new virtual machines online (e.g. following the concept of *immutable servers*).

Writing a deployment script therefore requires extensive knowledge of the platforms the application will be run on, the configuration of the servers, and the distribution of traffic. In order to minimize the effort required for creating such scripts, all companies standardize on a common deployment method. Within the interviewed group Docker [Merkel, 2014] is emerging as a defacto standard for application deployment.

5.4.4 Monitoring

As described in [Imamagic and Dobrenic, 2007] monitoring is often done on key components of software performance: measuring throughput, errors, or system loads. When concerned with microservices this approach is still valid, but does not longer convey all required information. In case a certain key system is unavailable it might affect lots of other systems as well. These systems do not have to be completely unavailable, but may be unable to offer their complete set of features. Measuring this using general performance characteristics does therefore not yield actionable metrics.

Therefore some interviewed organizations (*C2*, *C3*, *C7*) opt to measure their systems performance by adding key business metrics. This can be *the global number of likes in a set time interval* or *the number of completed orders per hour*. In case these metrics do not align with expected values, one can look into technological reasons for this (e.g. subsystem failure). As indicated by organizations which employ this metric, there are several additional benefits:

- The IT department is a lot closer to the business department in terms of shared goals,
- The metrics are better relatable to user goals,
- The metrics can also indicate the effect of qualitative changes (e.g. a user interface change).

A final key component is the monitoring of shared subsystems. As described earlier, many organizations opt for a central message bus. Since each service is free to publish messages to this bus, without any fine-grained control, such a system can easily become overloaded or ineffective. However in order to minimize cost and delays, such a key component of the organization should function without problems. Chapter 6 will expand on the monitoring of such message buses.

Chapter 6

Kafkalytics

With the understanding of microservices of Chapter 2 and 5, we can now focus completely on the communication between microservices. In this chapter the main focus will be on Kafka, a distributed message bus.

As described in Section 5.4.4 effective monitoring of shared systems is crucial in a microservice based architecture. Most of the interviewed companies employ a shared message bus, over which events are emitted if a certain action takes place. Since central governance for such is bus is not required, it is possibly an easy target for pollution. This could lead to reduced bus performance, or increased operating costs.

In order to solve this the authors developed *Kafkalytics*. One part of this system hooks into the production bus (in this case Kafka). The other part is embedded within the services which consume from the bus.

6.1 Kafka

Kafka [Kreps et al., 2011] is a message broker system, originally developed by LinkedIn and developed in the programming language Scala [Odersky et al., 2004]. LinkedIn decided to open source the project in 2011. The project was subsequently adopted by the Apache Software Foundation [Foundation, 2015]. In 2014 some engineers from LinkedIn founded a company (Confluent) which was dedicated to support the Kafka system.

The design of Kafka was strongly influenced by the way LinkedIn worked at that time. It was designed to be able to act as a unified platform for handling all the real-time data feeds a large company might have. The developers of Kafka state in the official Kafka design documentation the following:

- It would have to have high-throughput to support high volume event streams such as real-time log aggregation.
- It would need to deal gracefully with large data backlogs to be able to support periodic data loads from offline systems.
- It also meant the system would have to handle low-latency delivery to handle more traditional messaging use-cases.

- We wanted to support partitioned, distributed, real-time processing of these feeds to create new, derived feeds. This motivated our partitioning and consumer model.

- Finally in cases where the stream is fed into other data systems for serving, we knew the system would have to be able to guarantee fault-tolerance in the presence of machine failures.

The system is setup as the traditional *Pub-Sub* pattern (described in Section 2.2.2), although it abstract over *queueing* paradigm [Eugster et al., 2003] as well. The system has a Producer API, and a Consumer API. The producers connect directly to the brokers (the instances of a Kafka bus), whereas the consumers connect using a Zookeeper (a distrubuted coordination system) instance.

The bus itself is divided into one or more topics (configurable). A message can be sent to one or multiple topics at once. Each topic is further divided into partitions. The partitions allow the message logs to scale and allow for parallelism within the system. The consumers effectively consume entire topics from a number of partitions. The consumer decides client side which messages are of interest to it. The implicit contract of a Kafka consumer can therefore be defined as *consumption implies handling or ignoring*. Consumers can consume messages from the start of a topic or from any offset of the start.

Kafka is considered as a high performance, distributed, and scalable message bus. It has various use cases e.g. service communication and log data collection. It has a widespread use in the industry and is used by e.g, LinkedIn, Netflix, Spotify, Goldman Sachs, and others [Rao, 2016].

Kafka vs Enterprise Service Bus It may seem that Kafka is similar to the earlier concept of an ESB (as described in Section 4.3). However there are several key differences.

Kafka is in essence a so-called *dumb pipe*: it does nothing intelligent with the events it processes, except for handling parallelism and fault-tolerance. It does not do any message transformation nor routing. While an ESB can behave similarly to Kafka (featurewise), in practice it often behaved as an *intelligent pipe*. These paradigms are quite different: Kafka assumes all intelligence with regard to the messaging is part of the software systems (*smart endpoints*), while the ESB (often) encapsulates the intelligence in the bus itself.

As a result there are some features of an ESB not offered by Kafka. Examples are:

- Security,

- Encryption,

- Auditing,

- Routing,

- Message transformation.

For some organizations the lack of security, auditing, and encryption can be a setback. For example opening up Kafka for partners of an organization would mean complete access to all topics and messages in the bus. Following the paradigm described in the design docs, one should create an endpoint for these partners. These specific endpoints can push messages to the bus or forward messages to the partner.

A similar scenario holds for the latter two features. Routing should be handled by each application to determine whether it is interested in a certain message of a certain topic. Transformations of data should be done in the consumers itself (optionally before handing the message to the processing logic).

6.2 Kafka antipatterns

One of the key concepts of effective use of microservices is how developers *can be in control* (Section 2.3). Although this applies in many different parts of the ecosystem, it also means developers should be free to:

- Publish new types of messages to the bus system,
- Switch messages between topics,
- Consume any message from any topic from the bus.

The amount of freedom a team gains can therefore be large but it may come with a price. Once a message is within Kafka, each consumer of that topic will fetch the message, and then decide whether to do anything with it. It is therefore not possible to detect some problems at a bus level itself. Because any type of message can be transported (XML, JSON, or binary) the bus cannot infer any details based on the message content. Therefore the filtering takes place at the consumer level. Thus any message which is later filtered by a consumer is counted as delivered by the bus itself. Since there is lack of proper tooling in this regard (as far as the author is aware) this may eventually lead to code with antipatterns in it.

Dead messages There may be messages published in which no consumer will ever have any interest. Since one cannot infer this from either the IDE nor Kafka itself, this message will continue to be published by the producer. Effectively this producer is therefore a variant of dead code: *it does not affect the program results* [Kennedy, 1979]. Dead code in turn can lead to a variety of maintenance issues. Additionally these messages are still pushed to the Kafka bus, consuming resources and thereby making the bus less efficient. Finally these messages are still consumed and immediately disregarded by any consumers of the topic the message was published to. This leads to wasted resources on the consumers side, possibly requiring overly large resource allocations to the consumer. The removal of such messages may therefore lead to a lower load on the system as a whole. Since the dead code (*dead messages*) did not affect the program results, there are no downsides to this approach.

Strict interest Another downside may be that certain consumers are only interested in a strict subset of messages which are published to a topic. In order to consume this

set (which may have specific characteristics) every message published to that topic needs to be consumed. Consuming messages which the consumer is not interested in leads to wasted resources. Instead it can therefore be useful to split a certain topic into two separate topics such that these consumers can achieve a higher percentage of useful messages. However increased splitting of messages across topics may make it more difficult for developers to determine which topic to publish to, and to discover topics with useful events.

The balancing of messages over different topics therefore should be done carefully (not too much on one topic but not too many topics either). This requires a developer to strike a balance between decoupling of consumers and producers (not tailing data to the consumers needs in a non-reuseable way) and reducing operations costs (or increasing performance).

Similar topics On the other hand of strict interest we may have the opposite antipattern. Some consumers might need to subscribe to multiple topics for their messages. If multiple consumers consequently subscribe to a fixed collection of topics, it might be beneficial to publish the messages of these topics to a centralized topic. Later on, the other topics can be abandoned altogether. This in turn can reduce the number of partitions required to run Kafka (which in its default implementation is limited by the design of Zookeeper).

Write-only topics Since messages may be published to multiple topics, it can be the case that some messages are only read over a specific topic. The same messages published to another topic might never be read at all. However due to the design of Kafka, these messages are still saved to disk on the Kafka brokers and therefore consume both space and I/O. Since these topics are never read, their removal frees up resources for actively read topics. The avoidance of this therefore ensures lower operating costs of the bus system.

Message lag Kafka does not give any guarantees on the maximum delay which can occur between the production step of a message and its actual consumption. From the design documentation it follows that this delay is generally governed by the load on the bus and the number of consumer threads available. However for some processes it can be an indication of problems in case messages are delivered much later than they were produced. For these purposes it can be beneficial to keep track of the time elapsed between producing and consuming the message. If this message lag becomes too high it can happen that users are notified by email a long time after completing an order. This may then result in reduced customer satisfaction or more support requests.

Replayed messages The Kafka consumer contract in essence is the following: *once a message is consumed (committed offset) the consumer should either have processed or purposely ignored the message*. As a form of fault tolerance, some systems instead opt for a different approach. In case of an error, they place the current message back onto the topic they consumed it from in order to process it later. This approach however does have some significant downsides. First of all any other consumers of the topic may process the message twice, leading to unexpected results. In case there is

just a single consumer active on the the topic, this does not occur, but it has created the implicit constraint only that specific consumer can ever listen to the topic. This highlights an assumption made of the global state of all services in the system. Since this constraint cannot be defined in Kafka itself, it may be violated at a later stage, leading to double message consumption. Instead in case of an error a consumer should commit the last successful message to Kafka and stop. Upon starting again, it can continue with the message it was unable to process the last time.

Vast messages A producer can push messages to be bus of any size (which is configurable but defaults to 1MB). Such a message can include all sort of data, as determined by the producer. However large messages may take longer to parse and handle by the consumers, apart from requiring more resources from the Kafka brokers itself. In case a publisher creates messages which contain many unused fields, these fields may be removed at some point. This can in turn reduce the message size which has a positive impact on the operating costs of the bus. Depending on the exact needs (throughput in data size or in message count), larger or smaller messages may be beneficial [Kreps, 2014].

6.3 Kafka message monitoring using Kafkalytics

In order to function as a useful tool, Kafkalytics should be able to detect these antipatterns in an environment. To do so Kafkalytics will analyze the traffic over the Kafka bus. Services will then indicate to Kafkalytics which messages have been handled or ignored.

To evaluate Kafkalytics, we define the following goals:

- It should be able to detect all antipatterns mentioned in Section 6.2,
- It should allow for a gradual adoption within an existing infrastructure,
- It should be able to function within existing infrastructures without requiring any scaling,
- It should be easy to implement in existing services.

Analysis methods There would be two classical ways of analyzing the data in transit on the message bus: static and dynamic analyses. Static analysis could be provided to help a developer directly from within the IDE or provide feedback from a CI suite. This would yield shorter feedback loops, making it easy to integrate within the work flow of a developer or the team. However there are some problems with static analysis for this specific usage.

- Within a microservice centric organization many different technologies can be used, for each of which there has to be an analyzer available. If a single service cannot be analyzed, the analysis of the entire set of services might be off.
- Static code analyses cannot reveal how many messages are actually triggered since this depends on end-user input.

- Static analysis cannot detect code which is never called (which also depends on end-user input). This code is not dead code, but in production it might be code which is never executed.

- Most developers work with a set of fixture data, not with a database which represents production data. This may significantly change the performance characteristics of a system.

Since each of these issues would significantly reduce the effectiveness of the design, dynamic analysis was chosen. This analysis is performed in the live production environment of the system, so the actual usage counts can be inferred. However there are also a number of pitfalls for dynamic analysis which should not be ignored (at least one should be aware of them).

- Services need special code to report data back to another system,

- There is some special infrastructure required for reporting,

- Additional systems might need to be connected to the central bus.

In order to minimize the impact of these downsides, the design was carefully crafted in order to mitigate these downsides as much as possible.

Design The Kafkalytics systems consists of three main components. Firstly there is the *kafkalytics-server*. The second service is the *kafkalytics-api*. Finally there is a service called *kafkalytics-ui*.

Additionally there is a need to uniquely identify messages passing through the Kafka bus. Each message should therefore contain a unique identifier, and a date time when it was originally produced.

The Kafkalytics server is a service which will listen to Kafka topics (which once is configurable) and saves the data it received to a persistence layer. It setup should be such that it would be possible for multiple instances to run in parallel in order to capture all the traffic going through the bus. Additionally the system should be setup such that only one additional system needs to connect to the message bus system in order to reduce operational overhead.

For every message which a service received and deemed of importance, the identifier of the message needs to be saved. In order to keep development and operational effort low, this should leverage existing systems as much as possible. Moreover it would be beneficial for developers if the amount of boiler plate code would be reduced to the absolute mininum. This would allow them to use Kafkalytics in their services, without the required code obstructing the main goal of their work. The Kafkalytics API clients should not have a significant performance impact on the service itself.

Finally the Kafkalytics ui client is a web client, which allows the developer/operator of the organization to request certain insights on the message bus. These insights are visualized in appropriate graphs. Apart from performance characteristics, it may also highlight potential bus problems.

Implementation requirements In order to facilitate the usage of Kafkalytics within a system, a number of requirements are present. Based on the earlier interviews, these requirements seems reasonable for some organizations. With future development some of these requirements can be lifted.

- One should use Kafka,
- Ones messages should be JSON encoded,
- A message should include a message metadata object.

A message metadata object has the form of the JSON object presented in Listing 1 and should be present in the root of the message. In case a message does not conform to this format, it will be ignored by Kafkalytics. The resulting Kafka message should ultimately look similar to Listing 2.

```
1 {
2   "id": "unique message id",
3   "datetime": "iso8601 formatted date"
4 }
```

Listing 1: Message metadata object

```
1 {
2   "message": {
3     "id": "unique message id",
4     "datetime": "iso8601 formatted date"
5   },
6   "someNumber": 7,
7   "someString": "...actual message content here instead of
    ↪  the someNumber and someString dummy keys..."
8 }
```

Listing 2: Resulting Kafka message which can be processed by Kafkalytics

Implementation Kafkalytics-server runs with two different parts. One of these listens to all regular production topics on the Kafka cluster (configurable) and persists the items in bulk. The other listens to the special Kafkalytics topic (configurable) to which the kafkalytics api clients send their usage data. This data is also persisted using bulk inserts. This service is developed using NodeJS and for operator convenience runs within a self-contained Docker container.

The Kafkalytics-server uses an *Elasticsearch* [Gormley and Tong, 2015] document storage system with a single index and two types. A main reason for this was that this system was already available at the interviewed companies of Chapter 4, which would aid in the ease of adoption. This removes the need for additional special infrastructure.

In case a message flow is larger than a single server instance can handle, Kafkalytics can use the default Kafka design in order to load balance the messages over multiple instances.

The second component is an API client for this server component. This is the piece of code which is embedded within each microservice. In order to keep infrastructure changes to an absolut minimum, the API clients communicate with the Kafkalytics server using Kafka over a special topic (configurable). Since Kafka is already present for Kafkalytics-api to be useful, this prevents additional dependencies to be included in the project (which could cause conflicts). To reduce load on the services under measurement, the Kafkalytics will automatically cache results and flush these in bulk to Kafkalytics.

The API client has to be constructed using the same parameters required for connecting to Kafka itself:

- The name of the service to use for identification purposes *(required)*,
- The location of Zookeeper *(required, known for Kafka)*,
- The Kafka clientId *(required, known for Kafka)*,
- Any Zookeeper connection settings *(optional, known for Kafka)*,
- The Kafkalytics logging topic *(required, should be the same across all applications)*,
- The threshold for force persistence *(optional, default 100)*

When closing the application gracefully, Kafkalytics should be closed after the Kafka consumers so any remaining items in the queues can be saved. On each message which is deemed useful by the service, it should call the `send` method. This method takes the message id (present in the message metadata object) and the topic used for consumption (known in the consumer itself). The method is non-blocking and saves the message to the queue of the API client. A final method available in the API client is forcing a cache flush. This can be done when the instance is shutting down (this flush is not applied automatically, since it depends on the instance shutdown sequence) The earlier described kafkalytics-server listen to this special topic and saves the usages of messages.

The final component is the *kafkalytics-ui*. This uses the data saved to visualize the performance characteristics of the Kafka setup. Additionally it gives an overview of possible problems, as outlined in Section 6.2. It does not depend on Kafka, and just requires a connection to the persistence layer. The service is developed using NodeJS and also run within a self-contained Docker container.

6.4 Antipattern detection

Apart from integrating within the infrastructure, it should also indicate all of the aforementioned antipatterns. In this section a description will be given how Kafkalytics will determine whether antipatterns are present

Dead messages Since Kafkalytics listen to all messages on the Kafka bus, it will receive the original dead message. After a certain while, Kafkalytics can determine no consumer has marked the message as handled. In this case, there is a strong suspicion this message may be considered dead.

It can later analyze multiple dead messages to determine whether a set of these messages have a common aspect. In this case it can mark a group of messages as dead.

Strict interest In case some services only consumes a strict subset of all messages of a topic, it may be beneficial to extract these messages to a specialized topic. This can improve consumer performance.

Kafkalytics is able to determine which consuming services are interested in which messages. In case they did not consider the message contents to be relevant, the message would not have been marked as handled by the consuming service to Kafkalytics. Combined with clique detection it can be determined whether different services have similar interest. This is the case when the cliques of the different services overlap.

Similar topics As discussed similar topics might be an indication that a number of topics might be better of in a consolidated topic. In this case Kafkalytics can again perform a clique analysis on messages which it received, similar to strict analysis. It can combine this with the fact that other consumers are not interested in these topics.

Once it detects both conditions hold, a number of topics may be marked as similar. These topics are then candidate for merging into a single topic.

Write-only topics Since each message carries a unique identifier Kafkalytics can detect that some identifiers are never read over certain topics. If these messages are read on other topics, these messages are not considered dead. The topic information can be used, together with the message consumption, to determine that no message is ever read over that specific topic. In this case it can detect that topic does not have any subscribers, and mark the topic as write-only.

Message lag Due to the design of Kafkalytics each message contains the date and time it was produced. Each Kafkalytics api client will also record the date and time of the consumption of the message. This data can be combined into a delay between the production and consumption of the message.

This data can be performed for all messages in a certain topic (where only consumed messages are considered). This yields statistics on the average delay of messages in that topic, but also the standard deviation and several percentiles. In case this exceeds a configurable maximum, Kafkalytic will flag the topic as delayed.

The same can be done for certain consumers. Kafkalytics saves the name of the consuming service, and determine how long the service requires to handle the message. Also for this case once this number exceeds a certain maximum the service will be flagged as lagging.

Both approaches can be combined to determine whether a certain topic and subscriber have lags in their message handling. This will also flag the combination of the topic and the service as lagging.

Replayed messages Once Kafkalytics sees a message with a certain identifier twice on the same topic, the invariant of unique message ids is violated. This is an indication that a service has replayed a message to the topic.

If may also be possible that an operator has manually changed the offset of the service so the service actually reads the same message twice, without reproducing it. Since this has required manual intervention, this case will not be handled by Kafkalytics.

Vast messages Since each message is saved by Kafkalytics, it can compute statistics about the sizes of the messages passing through the bus. This includes the average message size, the maximum size, standard deviation, and several percentiles.

This data can be aggregated as a whole, or on a per topic basis. It is also possible to determine these statistics per consuming service or as a combination of topic and service.

Chapter 7

Evaluation

Now the goal and design of Kafkalytics are clear, we evaluate the system. In order to do so, we list the research questions of this thesis.

First we focus on the antipatterns which Kafkalytics can detect. The participants of Chapter 5 reflect on these antipatterns and indicate the seriousness of each antipattern. Secondly we evaluate the ability of Kafkalytics to detects these antipatterns. Next we look at the implementation overhead which is introduced by Kafkalytics. Finally we discuss the performance overhead of adopting Kafkalytics on a live system. We conclude the chapter by discussing any threats to the validity.

7.1 Research questions

The following research questions formed the basis for this thesis. They will be used to evaluate the information obtained from the interviews, the selected antipatterns, and the design and usefulness of Kafkalytics itself.

- To what extent do experts recognize the antipatterns described in Section 6.2?
- How well does Kafkalytics detect the aforementioned antipatterns in a production environment?
- How did developers perceive the implementation overhead associated with adding Kafkalytics to their existing system?
- What is the performance penalty of adding Kafkalytics to an existing system, compared to regular operations?

7.2 RQ1: To what extent do experts recognize the antipatterns described in Section 6.2?

As described in Chapter 6 several possible problems were mentioned when using a Kafka-like message bus as a central means of communication. These problems do not necessarily have to occur within every organization using the system.

In order to evaluate the seriousness of these antipatterns, we held talks with the organizations from Chapter 5. These participants of the interview have studied the

design and analyzed how their systems may suffer from the described problems in Chapter 6.

Within *C2* itself there is no instance of Kafka available, but the concepts underlying Kafka are well-known to the interviewed engineers. For these engineers the concept of dead messages is well-known and its impact to the system as a whole is understood. Both *C3*, *C4*, and *C7* have implemented actual Kafka buses within their organization. The participants from these organizations are therefore very familiar with the systems design. Additionally the problems can actually occur in their systems, which sparked additional interest since knowledge about possible problems can help to effectively mitigate these.

Dead messages As stated by participants from *C2*, dead messages may lead to potential long term technical debt. Over time the knowledge on the refactoring which removed the messages diminishes. If the production of such messages is not dealt with at an early stage, the message producer is less likely to be altered. A producer which produces dead messages is therefore more likely to keep producing these messages. Participants from *C3*, *C4*, and *C7* gave similar views on this topic. According to them, having automated tooling in place in order to detect this, can have added value. An important side note made by *C7* is that this tooling cannot be considered to hold an *absolute truth*. Instead the tooling gives an indication of a possible problem. It can for example be possible that during a certain timeframe a message may be considered dead, while the subscribing system is undergoing maintenance. It may also be the case a service was decommissioned before the new system was live (which is possible for non-realtime systems). A new system which started running does not handle those messages yet, but may do so in the near future. Removal of such messages is undesirable.

As a result of the above statements, the antipattern dead messages is considered to be an actual antipattern in Kafka. This antipattern has the possibility of introducing long term maintenance issues and is therefore considered serious.

Strict interest, similar topics & write-only topics The problem of strict interest of consumers was already known by engineers of *C3* and *C7* which had already encountered it. Although the parsing of JSON encoded messages can be done very fast, the delivery of all these messages to lots of non-interested consumers does mean an additional load. One of the raised points was that this also makes it much harder to hook into a production bus for debugging reasons due to the vast quantity of messages going through some topics. If certain systems are therefore only interested in a set of all messages delivered on a topic, it may be beneficial to create a separate topic. In order to not change existing consumers, one can publish the same message to multiple topics. Due to the setup of the systems in use at *C3* and *C7*, where messages already carry a unique identifier, the double delivery is therefore not a problem. It allows a system to start consuming two topics for its messages and gradually switch over the newly created topic.

An important side note in this regard is that it is not considered best practice by all interviewees to tailor the exact topics a message is published to to consumers. This could lead to coupling from producers to consumers, or highly specific usage patterns.

Additionally it ties different teams together, which according to Conways Law, may have an impact on the systems design. Instead the topic setup should be done as generic as possible with a clear division between the messages going over different topics.

The latter point also indicates the extent of the *similar topics* problem. Due to the fact having separate topics tailored to consumers is not considered a good practice, it rarely happens in production that therefore multiple topics process a large quantity of similar messages. *C3* does not have any messages present which are sent to multiple topics at all, whereas for *C7* this only happened on occasion. The latter does not intend to keep such practices around for long.

Due to the nature of the absence of such topics, the theoretical problem of *write-only topics* is not experienced by any of the interviewed parties. This indicates that developers are aware of the possible problems of having too strictly defined topics within their message bus infrastructure. However both parties embrace the notion of elimination of such topics from the system, in case they are inadvertently created.

An important side note made by *C7* is that due to the age of the system many possible problems do not occur because the system is not *old enough to have acquired this amount of technical debt*. To compare this with a regular monolithic structure *developers often also do not try to create a highly-coupled piece of software, but it may evolve into just that over time*. Since not enough time has passed since the initial conception of these systems, the amount of technical debt in this regard is relatively low. *C12* agreed that their current experience with technical debt using microservices and message buses is not yet enough to have a clear understanding of what may or may not happen in the (near) future. While the above problems are currently not present, this may change over time. As stated: *In the past none of our systems ever was designed to become more coupled over time. However when time passes this has happened multiple times*.

This evaluation suggests that these antipatterns may be considered present, but they pose little risk to systems. In case the antipattern is present in an application, an earlier antipattern (to tailor data to the needs of consumers) is already present. We therefore consider these antipatterns valid, but less applicable.

Message lag Message buses such as Kafka are not suitable for communication which is required to be real time (which is why *C10* does not employ such buses). There may be slight delays in the delivery of messages (in the order of microseconds). Under certain circumstances, where a producer produces messages faster than a consumer can handle them, this delay may grow significantly.

For all parties which have been interviewed and use message buses, a short delay is not considered to be an issue. However it is deemed beneficial by *C3*, *C4*, *C7*, and *C12* to measure the time which elapsed between the production and consumption of the message. Whether a delay happens due to problems on the Kafka side of the system or the consumption of the messages is not considered an issue itself.

Within organization (*C12*) it was also pinpointed that many metrics were saved for performance analysis or root cause analysis. Having accurate timings of how long messages were in the message bus waiting for processing was not one of these metrics. Since it was considered to be of importance to have easy and continuous insight in these numbers, plans are being made to employ a system which is able to keep track

of this.

We therefore conclude that reducing message lag can be important, but message lag itself does not necessarily pose a problem. Instead the measurement of message delays can prove beneficial (as *C3* demonstrated).

Replayed messages Replayed messages are considered a serious problem by *C3* and *C7*. These organizations depend on the defined characteristics of Kafka (at-least-once delivery) combined with an implementation which (consumer side) marks messages as processed after their processing completed successfully. However this implementation is consumer specific, hence in case another consumers re-produces the message, this global invariant is violated.

We will consider the following example to highlight the issue with replayed messages. Consider a topic where changes to entities are broadcasted. At some point an entity is changed, and this change is sent over the topic. For some reason the consumer crashes while handling the message, and re-produces it to the same topic. However before that re-produced message is handled, an entity deletion message comes in for the same entity. The entity is now deleted from the consumer. At some later stage, the re-produced message arrives and does no longer have an entity to change. In this case either the change is lost (which may be required for auditing) or the consumer might crash again due to the absence of the entity.

Concerning *C12* repeated messages is not considered a serious drawback, since their messages only have idempotent side-effects. However this does violate the assumption on the ordering of messages and therefore still considered to be risky under certain circumstances. The organization was not able to verify whether their systems depended explicitly on this ordering. *C12* therefore indicated that replayed messages are undesirable and this should be monitored.

Hence this antipattern is considered to be valid since all parties agreed on its validity. It can also be shown that this antipattern can directly have problematic effects, and might even violate legal requirements.

Vast messages As indicated by all parties, vast messages do not relate to actual code base problems. It may be required that messages have certain sizes due to the amount of information has to be embedded. Organization *C7* stated it might be useful to keep track of the message sizes over time. *C12* indicated that large messages may reveal a problem (lack of an API to query data, or messages are too generic). However these problems should not be addressed at a bus level. None of the organization has vast messages in their code base.

This proposed antipattern is therefore refuted since it is not considered to be of importance.

7.3 RQ2: How well does Kafkalytics detect the aforementioned antipatterns in a production environment?

To evaluate the detection of these antipatterns we asked the participation organizations whether they were willing to integrate Kafkalytics in their production environ-

ment. Organization *C3* agreed to this, and implemented Kafkalytics. Specifially the kafkalytics-server was ran (using two instances) on a small separate virtual machine. A total of 7 consuming services was equipped with the kafkalytics-clients. An Elasticsearch index was created on the primary Elasticsearch database cluster which consisted of three nodes. We then ran the experiment to analyze the bus traffic for 2 months. In this time period, Kafkalytics handled a total of 750 thousand messages which were produced. Additionallly 400 thousand consumption messages were generated by the Kafkalytics clients. Only systems which were developed inhouse by the organization were equipped with the system.

After running for two months, Kafkalytics had detected the following within the system:

- Some dead messages were found. One set of dead messages shared the `"class": "OrderProduct"` which was not handled by the system. Other groups of dead messages were also found.
- A problem was found regarding strict interest.
- No problems were found regarding similar topics or write-only topics.
- Some types of messages lagged significantly, and only during night time low traffic periods the system was able to catch up with the flow.
- No vast messages were detected.

Dead messages When dealing with the dead messages, some groups of dead messages were found. These messages shared a common factor (e.g. `"class": "OrderProduct"`). After a investigation by the engineers of *C3* it turned out this were entity changes which were broadcasted. Since these were also broadcasted for classes which happened to be implementation details for the producing service. As a result, the production of these entity changes was disabled for all classes which were not public for other services. This detection enabled the organization to better apply the concept of information hiding [Petitcolas et al., 1999].

The messages which were flagged as dead were actual dead messages. Kafkalytics did flag these messages correctly and was able to construct patterns on any similarities between these messages.

Strict interest At some point, Kafkalytics also flagged a topic as having the strict interest antipattern. Engineers of *C3* did not expect this, as they indicated at an earlier stage this antipattern would not be present within their organization.

Upon investigation, it turned out that multiple consumers all subscribed to the entity changes topic. Here they generally listed to all types of entities, except entities considered internal (as described in the previous paragraph). Kafkalytics therefore flagged the set of messages the consumers did listen to as strict interest, and recommended the extraction to a separate topic.

In this case Kafkalytics did not deliver. The actual problem was a number of dead messages on the topic. This automatically led Kafkalytics to mark the non-dead messages as a case of strict interest. Instead Kafkalytics should exclude the dead

messages from its strict analysis method. This may also indicate that not all analyses can be ran in isolation, since some analyses may be influenced by analysis of other antipatterns.

Message lag Shortly after implementing Kafkalytics in their infrastructure, *C3* received a warning about message lag on one of their topics. In this case the rate of message production during the day was significantly higher than the throughput provided by several consumers. During the night time, the consumers were then able to handle the load once the rate of new messages went down. Earlier assumptions of engineers of the company determined that some parts of the infrastructure could take some time (due to eventual consistency [Vogels, 2009]).

Using Kafkalytics the engineers were able to pinpoint the actual issue. Whereas it was assumed that a distributed database could take some time to become consistent, it turned this was not the problem. Instead it took some time before a message arrived at a consumer and was processed. Earlier optimizations to enable faster consistency in the database layer therefore proved to be completely ineffective. Using the Kafkalytics system, developers could hook into the event bus, and pinpoint the average delays over time (and other statistics such as the standard deviation). In the end this gave the insight that the number of instances of a certain service could be increased to mitigate any delays.

For this antipattern Kafkalytics did its job well. The first warning was sent 36 hours after the initial implementation in the subsystem service. Engineers were subsequently able to fix the issue within a few hours.

Conclusion Although *C3* considered the detectionto be effective, kafkalytics-ui was a heavily critized component. Due to way the system was originally built, some analyses take very long to complete. It can therefore not give a real time overview of the state of the message bus, but has to be run as a background job.

We conclude that Kafkalytics detects most described antipatterns well in a production environment. The system can be improved by ensuring kafkalytics-ui becomes faster and by coupling the results of some analyses.

7.4 RQ3: How did developers perceive the implementation overhead associated with adding Kafkalytics to their existing system?

The implementation overhead of Kafkalytics consists of several parts:

- Adding the kafkalytics server container to an existing infrastructure,
- Modifying the producers to comply with the message metadata format,
- Gradually switch services to use Kafkalytics,
- Adding the api clients to consuming services.

Adding the server container From experience of organization *C3*, this turned out to be straightforward. It required a small change in the server operations system (Puppet in this case). Due to the containerization of the server instance, it will run in isolation of other services. This ensures that deployments can be done quickly.

The code snippet of Listing 3 was sufficient to add the container to the existing infrastructure. Additionally the `config.json` file needs to be adapted. The complete `config.json` file is shown in Listing 4 on page 55. The total required number of lines is less than 100, of which the `config.json` is based on a provided example file. The responsible engineer indicated that the addition of this system component was straightforward and could be done within minutes.

```
1  class kafkalytics {
2
3   file { '/opt/kafkalytics':
4     ensure => directory
5   }
6   ->
7   file { '/opt/kafkalytics/config':
8     ensure => directory
9   }
10  ->
11  file { '/opt/kafkalytics/config/default.json':
12    ensure => present,
13    source => 'puppet:///modules/kafkalytics/config.json',
14    mode   => '0600',
15    notify => Docker::Run['rogierslag/kafkalytics']
16  }
17
18  docker::image { 'rogierslag/kafkalytics:latest': }
19  ->
20  docker::run { 'rogierslag/kafkalytics':
21    image => 'rogierslag/kafkalytics:latest',
22    volumes =>
    ↪  ['/opt/kafkalytics/config:/opt/kafkalytics/config']
23  }
24
25 }
```

Listing 3: The amount of Puppet code required to run kafkalytics-server

Modifying the producer code In order for Kafkalytics to function, each message which should be monitored should have an embedded message metadata object as shown in Listing 1 on page 41 This metadata object contains two parameters: a unique identifier and the current date and time (formatted according to ISO8601). *C3* chose to use UUIDs as unique identifiers. Additionally the programming languages in their software stack are able to format datetimes to ISO8601 directly. Since the `message`

section of the message itself was not previously used, the addition of these fields to the message considered to be easy. A code example for this (in Ruby) is shown in Listing 6.

Since the addition of these fields could not trigger any other behaviour by the consumers, these additions could be done gradually.

Gradual adoption Although Kafkalytics will provide the best data once the entire system is using is the system, it can also be implemented gradually. Since *C3* prefers to refrain for large deployment, portions of Kafkalytics were deployed one-by-one, once a system needed a modification. This means that data sent to Kafkalytics may be inaccurate and that since some data may be missing, the resulting data cannot be used for measurements. However this adoption pattern allowed *C3* to gradually adopt the system. while the system was not completely functional, it did not interfere in any way with regular service operations.

Addition of API clients To finalize the adoption of Kafkalytics, the API clients for it should be added to the consuming services. For *C3* this meant mostly NodeJS services. Adding Kafkalytics-api to these systems did not require an additional dependency, since these services already depended on the Kafka communication dependency.

Within each service Kafkalytics needs to be instantiated. This took only a few lines of code as can be seen in Listing 7. Most of this code was also required to connect the service itself to Kafka. Since Kafkalytics exposes a hook for when its ready, this could be integrated in the startup sequence of the service.

Once a message is deemed of importance and handled, the system should notify Kafkalytics. It can do so using the code of Listing 7.

Developers responsible for integration of the API clients in their services described this to be relatively simple. Additionally the code which is part of the main application code (e.g. the sending of messages) does not get in the way of the actual business logic.

A note was made on the shutdown sequence: Kafkalytics does not hook into the shutdown sequence, and developers have to add this themselves. This was initially not done, which caused queued messages to get lost. After discovery of the problem, the correct `kafkalytics.close()` call was added to the shutdown sequence.

Conclusion Apart from the shutdown sequence of the Kafkalytics API client, the responsible engineers considered the api client to be easy to work with. It did not get all over their code, did not interfere with business logic, did not add additional dependencies, and was properly containerized.

We therefore conclude that Kafkalytics has a low implementation overhead for developers and operators alike.

7.5 RQ4: What is the performance penalty of adding Kafkalytics to an existing system, compared to regular operations?

In order to evaluate the performance penalty of Kafkalytics in production, we compare it to other system running in the production environment. In the time period Kafkalytics ran at *C3* it handled a total of around 1.10 million messages which contributed to a traffic on the bus of 1.074GB. On the backend, it stored a total of 14GB of data (due to the duplication for searching).

We compare this with the log traffic which at *C3* was also sent over the bus. On a daily basis, this is equivalent to around 1.74 million messages, which is a total of 1.33GB on average. On the backend, this requires around 1.93GB of data.

These numbers indicate that the resource consumption of Kafkalytics on a two-month basis is comparable to the log data per day. The increase in resource consumption when adopting Kafkalytics is therefore no larger than 2%.

The persistence layer does have different storage requirements. Kafkalytics stored in two months 14GB of data, whereas the logging system stored 115.8GB of data. The adoption of Kafkalytics therefore increased the storage requirements with 12%.

If one considers that more systems than just logging are active, these numbers become even lower. We therefore conclude that the operational overhead of Kafkalytics can be considered to be low.

7.6 Threats to validity

There are a number of threats to the validity of the results we mentioned in this chapter.

7.6.1 Sample size

The sample size of the group of organizations of Chapter 3 (15 organizations) is relatively low. Additionally these organizations share a number of common characteristics, which means their result may not be generalizable to the entire industry:

- The organizations sizes were either small or large, no medium size organization (eg. 50 to 250 FTE) were present,
- All organizations have at least one IT office in the Netherlands,
- The organizations were open to discussions which they did not profit from directly.

The set of organizations which employed microservices was even smaller, just 6 organizations were able to participate on this. This reduces the sample size even more.

The final evaluation of Kafkalytics was done with a single company, since other companies were unable to integrate Kafkalytics in their production systems. This originated through either legal, business, or techincal requirements.

The small sample size can be explained due the fact that the majority approached organizations were unwilling to participate and donate the require time whereas they would not see any benefit of this.

7.6.2 Replication

The first threat of these results is the replication of this study. Due to the required anonimity of the participants, it may well be that with a different group slightly different results with emerge.

The final evaluation of Kafkalytics within a single organization are representative for that organization. However one cannot assume that these number are generalizable to the industry as a whole. The operational overhead of the system may vary between different users, depending on how the system was originally build.

Finally the gathered data from the production measurements cannot be made public to the confidential nature of this data. More statistics can be made available upon request, but the data itself cannot.

```
{
  "zookeeper": {
    "address": "CENSORED",
    "clientOptions": {}
  },
  "kafka": {
    "clientId": "kafkalytics-prod2",
    "groupId": "kafkalytics2",
    "topics": [
      {
        "topic": "cud-events"
      },
      {
        "topic": "auditing"
      },
      {
        "topic": "view-events"
      },
      {
        "topic": "perf-metrics"
      },
      {
        "topic": "monitoring"
      }
    ],
    "loggingTopic": {
      "topic": "kafkalytics-prod"
    },
    "bufferSize": 26214400
  },
  "elasticsearch": {
    "host": "CENSORED",
    "port": "9200",
    "apiVersion": "1.7",
    "index": "kafkalytics-prod",
    "type": "message",
    "logType": "usedMessage"
  },
  "client": {
    "kafkaThreshold": 10,
    "esThresHold": 10
  }
}
```

Listing 4: The applicable config.json file

```ruby
message: {
  id: SecureRandom.uuid,
  datetime: Time.current.utc.iso8601(3)
}
```

Listing 5: Ruby code required for the creation of the message metadata object

```javascript
var Kafkalytics = require('kafkalytics-node');
var kafkalytics = Kafkalytics({
 service: 'CENSORED',
 zookeeper_address: config.get('zookeeper.address'),
 kafka_clientId: config.get('kafka.clientId') + '-CENSORED',
 zookeeper_clientOptions:
 ↪  config.get('zookeeper.clientOptions'),
 loggingTopic: process.env.PROD ? 'kafkalytics-prod' :
 ↪  'kafkalytics',
 threshold: 5
});
```

Listing 6: NodeJS code required for initializing Kafkalytics

```javascript
function messageProcessor(message) {
 var topic = message.topic;
 message = JSON.parse(message.value);

 if (ofImportance(message)) {
   kafkalytics.send(message.message.id, topic);
   // ..other logic
```

Listing 7: NodeJS code required for marking a message as useful to Kafkalytics. Only line 6 has to be added for Kafkalytics.

Chapter 8

Related Work

Antipattern detection in Kafka using Kafkalytics has some similarities with other research areas. First of all there is the detection of smells in code. Secondly we discuss the monitoring solutions available.

8.1 Code smell detection

Code smells are an indication something might be wrong in software code. In case a smell is present, it is useful to dig to see whether an actual problem is found. Locations with code smells are generally candidate for refactoring. When compared with Kafkalytics the most notable difference is the way of operating: instead of performing analyses in a production environment it does a static analysis over the source code.

A well known system for detect code smells is the DECOR & DETEX system [Moha et al., 2010]. This system generates its own code smell detection algorithms, and can subsequently be run over a software project to identify possible problems. Developers or architects can later use the results of this tool to repay the technical debt.

Another issue regarding code smells is the nature of the code smell. Some code smells may be the result of complex business logic and not of bad design [Sjoberg et al., 2013]. The impact of code smells on the actual design is studied in e.g. [Yamashita and Moonen, 2012] and [Marinescu, 2012]. These papers have developed a framework in order to evaluate the severeness of several code smells and to which extent these influence the maintainance costs of the application at hand.

8.2 Service Monitoring

Service monitoring is essential to the successful operation of a microservice architecture. There are many ways to monitor services.

A popular monitoring framework is Nagios [Barth, 2008]. It is able to handle both passive and active checks and integrates monitoring with alerting keeping historical overviews of data.

Recent research on the topic of service monitoring has led to new initiatives such as ECoWare [Baresi and Guinea, 2013]. These systems are able to collect, aggregate and analyze real time data from running services. The system provides dashboard and alerting based on this performance data.

Chapter 9

Conclusions

This chapter gives an overview of the contributions of this thesis. After this overview, we reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

9.1 Contributions

The key contributions of this thesis are listed below. We will describe each contribution in more detail.

- Interview outcomes on monolith architectures and SOA,
- Interview outcomes on microservice architectures,
- Identification of seven Kafka antipatterns,
- The Kafkalytics monitoring infrastructure.

Interview outcomes on monoliths and SOA As described in Chapter 4 it turns out from the interviews with industry experts that the monolithic architecture is still often used. Each of the 15 organizations has built at least one system with this architecture. It also indicated monoliths are often fast to design, there is a lot of experience with them, and facilitates efficient communication. However it was noted that there is a tendency for these systems to become increasingly coupled over time. Additionally these systems cope with rising maintenance costs and are often less effective at scaling or fault-resilience.

The Service Oriented Architectures are used less over time. None of the organizations was currently developing such a system, nor had plans to do so. Although the principles of SOA seem to remain valid, the implementation of SOA went the other way: favouring big systems over smaller systems. Examples of this are the often complex standards for communication and the cumbersome ESBs. Some architectures using SOA are now being refactored to a microservice architecture using small services.

Interview outcomes on microservice architectures As it turns out from Chapter 5 microservices seems to be adapted quickly. Whereas the concepts seem similar to SOA, the implementation is not. Instead of revolving around technical entities, microservice architects pick business entities to serve as a basis for their services. Since services are easier to build, this in turn reduces the development time of individual services, which gives organizations an competitive advantage. Apart from that operational activities such scaling and fault-resilience become easier to achieve. It turns out organizations employing microservices consider the complete automation of building to deployment to be of great importance. The microservice movement therefore is strongly related to the devops movement.

Although it is technically possible to adopt many different technologies (polyglotism), many organization instead limit the size of their technology stack. Developers are free to chose from this stack, but the organization is not willing to support any type of database or programming language in production. Major reasons for this are the lack of experience and the lack of flexibility this polyglotism gives. Instead organizations consolidate on a number of technologies and stick with those.

Many organizations adopt a central message bus to facilitate communication between services. Apart from that, RESTful interfaces are exposed to make data accessible for other services. Services can talk freely to each other, and share a common authentication and authorization layer.

However it has also turned out that there are some downsides present. A major factor is a refactoring across service boundaries. This will often result in multiple services needing to change, due to the communication penalty involved this become more complex. Finally organizations indicated they currently still lack exprience with technical debt in this architecture. They therefore do not (yet) know where and when issues will manifest themselves.

Identification of seven Kafka antipatterns In order to detect any antipattern on a Kafka bus when using microservices, we described 7 different antipatterns. These patterns are described in Section 6.2 and evaluated in Section 7.2.

The following antipatterns were identified:

- Dead messages,
- Strict interest,
- Similar topics,
- Write-only topics,
- Message lag,
- Replayed messages,
- Vast messages.

These antipatterns can either pose problems with future maintenance or the performance of the central bus or consumers. A combination of problems is also possible.

The Kafkalytics monitoring infrastructure In order to give architects and developers a tool to automatically detect these antipatterns in their production environment, we developed the tool Kafkalytics. Kafkalytics consists of three elements: a server, api client, and a UI component.

Kafkalytics has a low operational overhead and is considered to be easily implementable in existing systems. It can reliable, although slowly, run analyses on the aforementioned antipatterns and flag certain sets of messages, topics, or consumers.

9.2 Implications

The results of the interviews chapters (Chapter 4 and 5) give a unique insight in the usage of different architectural styles by the industry. These show how organizations handle architectural challenges and can be used to highlight differences between the industry and the academic world. Future researchers may be able to better tailor their research to the needs of the industry. This may allow for a better and more effective knowledge transfer between those two parties.

Secondly the analysis of different antipattern when a message bus is considered can give architects and developers better insight in best and worst practices. Organizations can benefit from this knowledge: if these antipatters are avoided, software may be less likely to develop technical debt over time. A lower amount of technical debt can then lead to shorter delivery cycles and less bugs.

Finally Kafkalytics gives architects, developers, and operators (although this can be one person in a devops team) adequate insight in the performance of the central component of their architecture: the message bus. They can effectively see and measure what is going over their bus, and optimize their infrastructure accordingly. Antipatterns are automatically flagged, after which they can decide whether this was a valid detection. If that is considered to be the case, actions can then be taken to remove the antipattern.

9.3 Limitations and future work

As stated in Section 7.6 there are some threats to the validity. Most of the future research which we suggest is aimed at eliminating these threats.

9.4 Interview phase

As discussed the group which did participate with these interviews was relatively small. It is therefore possible the results are skewed in a certain direction. The following suggestions are applicable to improve this phase.

Increasing the sample size The sample size for organizations mentioned in Chapter 3 was relatively limited. Especially regarding the evaluation, only 4 organizations were able to accurately evaluate the systems design. In order to determine whether some of the noticed trends and anecdotal evidence are common within the industry as a whole a larger sample group is required.

Several improvements can be made here:

- Geographical differences between organizations,
- Cultural differences between organizations,
- A larger variety in the main industries between organizations
- A larger variety between the sizes of organizations.

Re-interview As noted in Chapter 7 some organization indicated they did not have experience with technical debt in their microservice architectures yet. This was mainly due to the fact these architectures have not been around for long enough to actually start building up technical debt.

In order to evaluate how technical debt accumulates in such systems, the same type of talks could be held with organizations a few years from now. This would allow their architectures to evolve and build up some of this debt. At that point, the organizations are more likely to detect the debt. Additionally they will have first hand insights in how it may develop over time, and how it may manifest itself.

9.5 Kafkalytics

As a first prototype Kafkalytics was evaluated to do its job well. However there are some further enhancements to make.

Antipattern identification We do not claim the listed antipatterns are the sole antipatterns when using Kafka. In the future more antipatterns might be identified by research or experience. More antipatterns might lead to a better set of best and worst practices, which can improve the code quality. As a result of this, delivery times and likeliness of bugs can be positively impacted.

Extending Kafkalytics The Kafkalytics concept of following and tracing messages can be extended to other services. In this specific case, one does not necessarily have to assume such as system is a message bus. For example: the concept can also be generalized to email within a large multinational: which emails are of interest to who and with which delays to people act on them?

Other message systems can also benefit from the analysis patterns Kafkalytics can detect. Examples of such systems are RabbitMQ or ActiveMQ. Each of those systems could benefit such analyses. However one would need to evaluate whether for these type of messaging systems (whether email or message buses) the same antipatterns hold.

Improving Kafkalytics performance As discussed in the evaluation of Chapter 7 the performance of the kafkalytics-server and kafkalytics api clients are more than sufficient. Currently this is not the case for the UI components, since several analyses take long to complete. Therefore Kafkalytics is not yet ready to function as a realtime overview on a message bus.

In order to improve usability, parts of Kafkalytics might have to be rewritten to use a different persistence layer. The current implementation which uses Elasticsearch

cannot perform well on certain query types. A dual persistence layer might be beneficial where read queries can be executed to the layer which is suited best for the specific query type.

Bibliography

[Abbes et al., 2011] Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 181–190. IEEE.

[Alonso et al., 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). *Web services*. Springer.

[Balalaie et al., 2015] Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Migrating to cloud-native architectures using microservices: An experience report. *arXiv preprint arXiv:1507.08217*.

[Baresi and Guinea, 2013] Baresi, L. and Guinea, S. (2013). Event-based multi-level service monitoring. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 83–90. IEEE.

[Barth, 2008] Barth, W. (2008). *Nagios: System and network monitoring*. No Starch Press.

[Booch, 2006] Booch, G. (2006). *Object oriented analysis & design with application*. Pearson Education India.

[Buschmann, 2011] Buschmann, F. (2011). To pay or not to pay technical debt. *Software, IEEE*, 28(6):29–31.

[Cai et al., 2000] Cai, J., Kapila, R., and Pal, G. (2000). Hmvc: The layered pattern for developing strong client tiers. *Java World*, pages 07–2000.

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75.

[Chappell, 2004] Chappell, D. (2004). *Enterprise service bus*. " O'Reilly Media, Inc.".

[Conway, 1968] Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[De Jong, 2015] De Jong, M. (2015). *Zero-Downtime SQL Database Schema Evolution for Continuous Deployment*. PhD thesis, TU Delft, Delft University of Technology.

[DiCicco-Bloom and Crabtree, 2006] DiCicco-Bloom, B. and Crabtree, B. F. (2006). The qualitative research interview. *Medical education*, 40(4):314–321.

[Durham Goode, 2014] Durham Goode, S. P. A. (2014). Scaling mercurial at facebook. https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/. [Online; accessed 1-March-2016].

[Duvall et al., 2007] Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[Erl, 2008] Erl, T. (2008). *SOA design patterns*. Pearson Education.

[Etzion and Niblett, 2010] Etzion, O. and Niblett, P. (2010). *Event processing in action*. Manning Publications Co.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131.

[Fielding, 2000] Fielding, R. (2000). Fielding dissertation: Chapter 5: Representational state transfer (rest).

[Foundation, 2015] Foundation, A. S. (2015). Apache software foundation. https://www.apache.org/. [Online; accessed 30-December-2015].

[Gormley and Tong, 2015] Gormley, C. and Tong, Z. (2015). *Elasticsearch: The Definitive Guide*. " O'Reilly Media, Inc.".

[Hitz and Montazeri, 1995] Hitz, M. and Montazeri, B. (1995). *Measuring coupling and cohesion in object-oriented systems*. Citeseer.

[Hoff, 2010] Hoff, T. (2010). Netflix: Continually test by failing servers with chaos monkey.

[Httermann, 2012] Httermann, M. (2012). *DevOps for developers*. Apress.

[Imamagic and Dobrenic, 2007] Imamagic, E. and Dobrenic, D. (2007). Grid infrastructure monitoring system based on nagios. In *Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28. ACM.

[James Lewis, 2014] James Lewis, M. F. (2014). Microservices. http://martinfowler.com/articles/microservices.html. [Online; accessed 8-March-2016].

[Katzan Jr, 2009] Katzan Jr, H. (2009). Cloud software service: concepts, technology, economics. *Service Science*, 1(4):256–269.

[Kennedy, 1979] Kennedy, K. (1979). *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division.

[Krafzig et al., 2005] Krafzig, D., Banke, K., and Slama, D. (2005). *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional.

[Kreps, 2014] Kreps, J. (2014). Benchmarking apache kafka: 2 million writes per second. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-seco [Online; accessed 30-December-2015].

[Kreps et al., 2011] Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. NetDB.

[Leffingwell, 2007] Leffingwell, D. (2007). *Scaling software agility: best practices for large enterprises*. Pearson Education.

[Manabe and Imase, 1992] Manabe, Y. and Imase, M. (1992). Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15(1):62–69.

[Marinescu, 2012] Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, 56(5):9–1.

[Marsden, 2015] Marsden, L. (2015). The microservice revolution: Containerized applications, data and all. http://www.infoq.com/articles/microservices-revolution. [Online; accessed 2-March-2016].

[Martin, 2003] Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

[McConnell, 1996] McConnell, S. (1996). Daily build and smoke test. *IEEE software*, 13(4):144.

[McKendrick, 2015] McKendrick, R. (2015). Monitoring docker.

[Merkel, 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Moha et al., 2010] Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36.

[Newman, 2015a] Newman, S. (2015a). Backends for frontends. http://samnewman.io/patterns/architectural/bff/. [Online; accessed 11-February-2016].

[Newman, 2015b] Newman, S. (2015b). *Building Microservices*. " O'Reilly Media, Inc.".

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the scala programming language. Technical report.

[Parnas et al., 1983] Parnas, D. L., Clements, P. C., and Weiss, D. M. (1983). Enhancing reusability with information hiding. *Tutorial: Software Reusability*, pages 83–90.

[Perrey and Lycett, 2003] Perrey, R. and Lycett, M. (2003). Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE.

[Petitcolas et al., 1999] Petitcolas, F. A., Anderson, R. J., and Kuhn, M. G. (1999). Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078.

[Ranchal et al., 2015] Ranchal, R., Mohindra, A., Manweiler, J. G., and Bhargava, B. (2015). Radical strategies for engineering web-scale cloud solutions. *Cloud Computing, IEEE*, 2(5):20–29.

[Rao, 2016] Rao, J. (2016). Powered by. `https://cwiki.apache.org/confluence/display/KAFKA/Powe` [Online; accessed 12-June-2016].

[Rodríguez-Domínguez et al., 2012] Rodríguez-Domínguez, C., Benghazi, K., Noguera, M., Garrido, J. L., Rodríguez, M. L., and Ruiz-López, T. (2012). A communication model to integrate the request-response and the publish-subscribe paradigms into ubiquitous systems. *Sensors*, 12(6):7648–7668.

[Rotem-Gal-Oz, 2006] Rotem-Gal-Oz, A. (2006). Fallacies of distributed computing explained. `http://www.rgoarchitects.com/Files/fallacies.pdf`. [Online; accessed 12-Januari-2016].

[Rozanski and Woods, 2011] Rozanski, N. and Woods, E. (2011). *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

[Savir and Laurer, 1975] Savir, D. and Laurer, G. J. (1975). The characteristics and decodability of the universal product code symbol. *IBM Systems Journal*, 14(1):16–34.

[Siroker and Koomen, 2013] Siroker, D. and Koomen, P. (2013). *A/B Testing: The Most Powerful Way to Turn Clicks Into Customers*. John Wiley & Sons.

[Sjoberg et al., 2013] Sjoberg, D. I., Yamashita, A., Anda, B. C. D., Mockus, A., and Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *Software Engineering, IEEE Transactions on*, 39(8):1144–1156.

[Smith et al., 1998] Smith, R., Harrison, R., Wood, S., Sussman, D., Fedorov, A., Murphy, S., et al. (1998). *Professional Active Server Pages 2.0*. Wrox Press Ltd.

[Stephens, 2015] Stephens, R. (2015). Beginning software engineering.

[Tilkov, 2015a] Tilkov, S. (2015a). Don't start with a monolith. `http://martinfowler.com/articles/dont-start-monolith.html`. [Online; accessed 11-February-2016].

[Tilkov, 2015b] Tilkov, S. (2015b). The modern cloud-based platform. *IEEE Software*, (2):116–116.

[Turnbull, 2013] Turnbull, J. (2013). *The Logstash Book*. James Turnbull.

[Turnbull, 2014] Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull.

[Varia, 2010] Varia, J. (2010). Architecting for the cloud: Best practices. *Amazon Web Services*.

[Vogels, 2009] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1):40–44.

[Yamashita and Moonen, 2012] Yamashita, A. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315. IEEE.

Appendix A

E-commerce using microservices

In this appendix a fictional E-commerce site system will be described. This system will adhere to some of the best-practices which arose from Chapter 4. It has been designed in cooperation with *C3* of Chapter 4 and 5. For simplicity, a number of details has been left out.

A.1 Entities

The system consists of a number of entities. These entities revolve around business entities, which are used to communicate between departments of the organization. The choice to use business entities here was simple: terminology which is used between departments is unlikely to change, hence these models provide a good basis.

The following entities have been chosen:

- User
- Order
- Shop
- Product
- Payment
- Support
- Fulfillment
- Notification
- PDF

A.2 Systems

Almost each of the entities in converted to a service. All of these services are free to communicate with each other using RESTful communication, and each service can put messages on a Kafka bus.

Services can defer tasks to inner services, which are specific to a certain department. These inner services cannot be accessed from other services, except through the outer service.

In this section each of the services will be described in more detail.

User The user service is responsible for creating users and storing their personal information. It also handles authentication of users, resets passwords, and maintains the correct access rights. This service is therefore the only authoritative source of truth concerning this data

Order, Shop & Product Since our system has multiple shops (and orders belong to a shop) the system co-locates these three entities. This is an example where it can be beneficial not to promote each entity to a new service as this would result in overly chatty communication.

In this case an order belongs to a certain shop and a certain user. A shopping cart is modeled as an order which has the special status `open` (other statuses are `cancelled`, `expired`, `refunded`, `completed`).

A `Product` can be available in only certain shops, or it may have different pricing. Therefore an abstraction is made to `ShopProduct` which keep this data. In turn orders keep their contents using `OrderProducts`, which save the price of the product upon selection (and refer to a `ShopProduct`).

Payment Dealing with financial data is considered hard, hence the payment service is split up into several inner services. Each payment method is a separate service (due to the different legal requirements for such services).

The inner services of the payment service are:

- `IdealPayment`: this service facilites the Dutch payment method iDeal.
- `CreditcardPayment`: for legal requirements, this service handles credit card data, which is not allowed to leave the system after entering.
- `BitcoinPayment`: for technical reasons (Bitcoin speed) this system observes and handles the blockchain data.
- `PaypalPayment`: this service handles all PayPal related funds, including possible refunds and automated mitigation of any Paypal complaints.
- `WiretransferPayment`: for technical and legal reasons this system interacts directly with the acquirer bank of the company.

Fraud detection is part of the outer service. This section of the service checks (among others) the previous payment statuses of the customer, checks for support messages and the country the order originates from. It returns a score ranging between 0 and 100 how likely this order is fraudulent one. Depending on the exact shop the order is done in, the payment may be denied (or a certain `PaymentMethod` may be deemed unfit).

Support The support service receives calls from the `Notification` service (which will be described later). It enters an email as a new `SupportRequest` or appends the email as a `SupportReply` to an existing `SupportRequest`. It communicates heavily with the `User` and `Order` services to fetch and show relevant information to the personnel. Using the `Notification` service it can show all earlier (manual and automatic) communication with the customer.

Fulfillment Once an order changed its status to `completed`, it enters the fulfillment status. During this stage goods might need to be shipped, or digital downloads can be prepared. If an order moves from `completed` to `refunded` these downloads are automatically reverted, or a return policy is started. The service informs the user about the fulfillment process using the notification service.

Notification The notification service is responsible for communication with the end-user. It can send text messages or emails, and keeps track of the delivery status of these items, the click-through ratio, and some other parameters. For email, the service is responsible for embedding the HTML content in default company styling.

PDF The PDF service handles the creation of different types of PDF's. Examples can be e-ticket generation, invoice generation, or watermarked ebook downloads. The service accepts a template and a content body, which are combined using the ruleset of the template. Finally the service returns a binary PDF document to the requesting service.

A.3 Design

Following these services, the design compromises of several other patterns as well. These are communication, authentication & authorization, aggregation, and state.

Communication The different services can communicate with each other over HTTP using RESTful interfaces. Additionally each service can put messages on a shared Kafka bus. This bus has several topics on which each service may publish information:

- `cud-events`: Events when an entity in a system is created, updated, or destroyed.
- `view-events`: Events when an entity was requested through the API.
- `auditing`: Any events relating to security involved practices, password changes, impersonation, or handling financial data.
- `log`: All log messages are published to this topic to be picked up by log aggregators and saved in a ELK stack [Turnbull, 2013].
- `kafkalytics`: The Kafkalytic feedback loop.

Messages on the `cud-events` and `view-events` are strictly structured. Each message contains a message metadata object, the action done, the entity involved, the ID of the entity, the API URL for the entity, the API URL of previous version of the entity, and the change set. An example message on the `cud-events` topic looks therefore similar to the response as given in Listing 8.

```
{
  "message": {
  "id": "srv3__deploy_2016.01.11.06.33__
↪  6774c0e3-38d3-4696-80f9-8e57d360f6aa",
  "datetime": "2016-01-11T12:48:53.379Z"
  },
  "action": "update",
  "class": "Order",
  "id": 263516,
  "url": "/orders/263516",
  "previous_version": "/versions/1978217",
  "object_changes": {
    "status": [
      "open",
      "expired"
    ],
    "updated_at": [
      "2016-01-11T12:43:49.840Z",
      "2016-01-11T12:48:53.335Z"
    ],
    "lock_version": [
      0,
      1
    ]
  }
}
```

Listing 8: Kafka cud-event message

The embedding of the change set allows each consumer service to directly identify whether this message is of interest. In this example the service can directly see an Order with ID 263516 changed from status `open` to `expired`.

If required a service can request the exact previous state of the Order, which is saved as an immutable version and available through the API. Once this endpoint is requested with the proper credentials the API will return a response similar to Listing 9.

Here one can also see who changed the entity (`whodunnit`, `null` indicates a system action), the exact representation of the entity at the time, the IP of the requester (if any), the host performing the action (stripped in this example), and the sha of the git tag in use at the time on that host (also stripped in this example). These Kafka messages are automatically generated for each entity update (for specified entities) once the commit

```
{
  "id": 1978217,
  "item_type": "Order",
  "item_id": 263516,
  "event": "update",
  "whodunnit": null,
  "object": {
    "id": 263516,
    "created_at": "2016-01-11T12:43:49.840Z",
    "updated_at": "2016-01-11T12:43:49.840Z",
    "user_id": 288958,
    "shop_id": 123,
    "status": 3,
    "timeout_after": "2016-01-11T12:48:49.816Z",
    "fee": 0,
    "lock_version": 0
  },
  "created_at": "2016-01-11T12:48:53.335Z",
  "ip": null,
  "host": null,
  "git_tag": null,
  "object_changes": {
    "status": ["open", "expired"],
    "updated_at": ["2016-01-11T12:43:49.840Z",
↪ "2016-01-11T12:48:53.335Z"],
    "lock_version": [0, 1]
  }
}
```

Listing 9: Version API response

of the transaction is confirmed. This prevents race conditions where the change set in the message is computed incorrectly, or the message is sent before the transaction actually completed.

Authentication & Authorization Authentication between services and users (or services and services) is performed using user or machine tokens. The `User` service is the authoritative resource for all authentication related matters. All services perform automatic scoping for entities, hence some products may return a `404 Not Found` since these entities do not exist for the specified user (e.g. these products might be hidden for that user).

Aggregation To improve search speed and certain online analytics, Elasticsearch is used. Certain entities of certain services are automatically updated in Elasticsearch after a change. Elasticsearch is however never the source of truth for the system,

because the persistence to Elasticsearch is done in a asynchronous task and the system can therefore be slightly outdated (also see the next paragraph on state).

Elasticsearch allows for rapid graphing of key characteristics of the platform by performing aggregations on e.g. sales and requests. It also functions as the backend for developers in order to spot performance issues or errors using a Kibana interface. Finally Kafkalytics uses Elasticsearch as its persistent backend.

State All services should remain stateless themselves in order to facilitate easy load-balancing and fault-resilience. In order to make this possible no service is allowed to keep queues within the system. Pushing data to an external service (e.g. Kafka or a DBMS) should therefore happen within the context of an HTTP request or a task itself. In case a task should be performed, the service will save the task to a Redis instance. Redis is an in-memory data store, which in this case is used as a cache and message broker for certain actions. All instances of that service poll Redis for tasks and execute these. Once a task is started this is marked in Redis. In case the processing node goes down, the task is reassigned to another node. Once completed it is marked as completed within Redis.

This setup allows for asynchronous tasks to be executed without being in a special context. To mitigate the issue of stale results in Redis, only identifiers, classes, and tasks names may be saved. This prevents outdated models from being processed. For some entities (such as the one in Listing 9) a `lock_version` is saved as well. Once this attribute is updated (and the task is therefore non-fresh) the task should be re-created using a strategy saved in Redis as well. Examples of this are e.g. the timing out of orders. Saving an entity with a lower or equal `lock_version` will always trigger a `StaleRecordError`, which prevents concurrent overwrites of records in a DBMS.

A.4 Walk though of a successful order

In order to more effectively understand the structure of this system, we will walk through the system while imagining that an successful order is submitted. The system actions are described in the text itself.

Process The process starts when a user visits a web shop. Automatically a session is created. As no login information is sent along a new anonymous user is created as well. For every subsequent request, the session token is sent in the HTTP headers to identify the user doing the request. Using this session a new order is created, with status `open` and a timeout of 15 minutes in the future.

The user's browsers now requests the list of shop products available. By modifying the quantities, the browser sends updated Orders to the server. Each time the order is updated, a Kafka event is fired (with class `Order` and action `update`).

Once the user progresses to the next screen, he/she is asked to provide his/her personal information. After this information is filled in, this is sent to the server as well. The *User* service checks the data (and verifies it confirms to the business logic). If the update is allowed, another Kafka message is dispatched (with class `User` and action `update`).

In the final step the user can select a payment method to complete the order. Once the user has selected a payment method, a request is done to the *Order, Shop & Product* service to make the payment. This service does a call to the *Payment* service which performs a fraud analysis. If the possibility for fraud is low enough for the specific shop, the payment is allowed. The *Payment* service now calls the responsible inner service to perform the payment with the correct netto amount. This service prepares the payment with the external party, and returns a URL to the *Order, Shop & Product* service. This service then sends that URL to the user, which is redirected to that URL to perform the payment.

After the payment has completed the user is sent back to the *Payment* service. Depending on the external party, the payment status may have already been pushed to the service. If not, the payment status is fetched first. Assuming the payment was successful, the user is thanked and an order description is shown (which in turn was fetched using a GET request from the *Order, Shop & Product* service). In the background the *Payment* service has triggered a Kafka event over the *cud-events* topic indicating an update of the payment (to the status `successful`). The *Order, Shop & Product* service reacts to this by fetching the payment details and checking that the entire order has been paid. If this is the case, the order changes state to *completed* and the corresponding Kafka event is fired.

At this point the *Notifications* service decides it should send a confirmation email. It first fetches the user data from the *User* service and the order data from the *Order, Shop & Product* service. These are combined in a template and sent by email.

In the meantime the *Fulfillment* service has started working as well, based on the Kafka message. It fetches the order data from the *Order, Shop & Product* service and determines how the order should be fulfilled. In this case a physical product needs to be shipped to the customer, hence a packing slip is created for the warehouse. Once the package is completed and ready for shipment, the *Fulfillment* service sends a Kafka event. The *Notifications* service reacts to this event, and sends the customer a text message with a parcel tracking code.

Finally the *Order, Shop & Product* creates an invoice by doing a POST request to the *PDF* service. It saves the resulting PDF for data compliancy in the central data lake.

Notes This example point out some interesting advantages of decoupling and modularization of services using microservices combined with Kafka.

- The *Fulfillment* service may temporarily be unavailable without impacting the overall process. All packing slips and downloads will ultimately be dealt with.
- The same goes for the *Notifications* service. In case there are intermittent problems, an order confirmation email can be delayed, but this failure does not impact the warehouse in any way.
- The *PDF* is much more tightly coupled. In case of downtime, a final invoice cannot be created. The *Order, Shop & Product* service therefore schedules this as a new job using Redis (such that it can be done once the service is back).

- The *User* service is a central component, since it handles the central authentication and authorization. The availability of this service is therefore crucial. Due to the structure many instances of this service can run at the same time, such that host failure does not impact overall availability.
- Problems with a specific payment method do not impact any other payment methods. The outer *Payment* service can simply deny all payments requesting a problematic payment method and continue to function.
- All components can be individually scaled and depending on the exact business needs each can have a certain level of fault tolerance. During redeploys internal load balancers can instantly redirect traffic to new instances due to the lack of local state.