Transformation of a Monolithic Architecture into a Highly Scalable Microservices Cloud Architecture Based on an Existing .NET Application

Dominik Zöchbauer

MASTERARBEIT

eingereicht am

Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Dezember 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, December 19, 2019

Dominik Zöchbauer

Contents

Abstract

Existing web applications are often implemented following a monolithic approach. A fast growing code base continuously introduces greater complexity and will eventually reveal the technical and organizational limitations of a monolithic application. A solution to overcome these limitations is the transformation into a microservices architecture.

In a first step, this theses gives an overview of advantages and disadvantages of the microservices pattern and helps to decide whether a transformation from a monolithic to a microservices architecture is worthwhile or profitable. Based on an open source application implemented in .NET that serves as an example, an incremental approach for splitting a monolith into microservices is demonstrated and evaluated. Concluding, benefits and drawbacks of a microservices architecture are identified.

Furthermore, the created microservices are containerized using Docker. By utilizing container orchestration platforms, the microservices application is deployed onto cloud infrastructure. The available orchestration platforms in Azure are Azure Kubernetes Services (AKS), Service Fabric on Linux and Service Fabric on Windows. The evaluation of these platforms with regard to their functionality as well as performance shows that the Service Fabric platform is lacking mechanisms for full encapsulation and only provides basic functionality for container management. However, under high load, a Windows-based Service Fabric cluster performs significantly better than AKS and Linux-based Service Fabric.

Kurzfassung

Bestehende Implementierungen von Webanwendungen folgen oft einem monolithischen Ansatz. Eine schnell wachsende Codebasis bringt auch eine kontinuierlich steigende Komplexität mit sich und wird früher oder später die technischen sowie organisatorischen Einschränkungen einer monolithischen Anwendung aufzeigen. Eine Lösung zur Überwindung dieser Einschränkungen ist die Transformation in eine Microservices-Architektur.

Diese Arbeit gibt im ersten Schritt einen Überblick über die Vor- und Nachteile des Microservices-Patterns und bietet eine grundlegende Entscheidungshilfe bei der Frage, ob eine Transformation von einer monolithischen zu einer Microservices-Architektur einen Mehrwert bringt. Anhand einer beispielhaften Open-Source-Anwendung, die auf .NET basierend implementiert ist, wird ein inkrementeller Ansatz zur Zerlegung eines Monolithen in Microservices demonstriert und bewertet. Abschließend werden Vor- und Nachteile einer Microservices-Architektur herausgearbeitet.

Weiters werden die erstellten Microservices mittels Docker als Container bereitgestellt. Unter Verwendung von Container-Orchestrierungsplattformen wird die Microservices-Anwendung auf Cloud-Infrastruktur bereitgestellt. Die in Azure verfügbaren Orchestrierungsplattformen sind Azure Kubernetes Services (AKS), Service Fabric auf Linux und Service Fabric auf Windows. Die Evaluierung dieser Plattformen hinsichtlich ihrer Funktionalität und Performance zeigt, dass Service Fabric keine adequaten Mechanismen bereitstellt, um eine vollständige Kapselung von Bereichen oder Komponenten zu ermöglichen. Darüber hinaus bietet Service Fabric nur Basisfunktionalitäten für die Verwaltung von Containern. Unter hoher Last bietet ein Windows-basierter Service-Fabric-Cluster jedoch eine erheblich bessere Leistung als ein AKS-Cluster bzw. Linux-basierter Service-Fabric-Cluster.

Chapter 1

Introduction

As web applications grow in size and complexity, scaling these applications becomes more important. Reasons for this growing complexity of web applications could be an increasing customer or user base. This leads to a higher load of requests an application receives. To deal with such a high load, there are two approaches to scale an application: scaling up and scaling out – these are also known as vertical and horizontal scaling respectively. Scaling up basically means to add more resources to a server that will eventually reach an upper limit. In contrast, scaling out means to replicate the server and run these replications in parallel. Since there is no hardware limitation with this approach, it should be preferred in the long run.

An architectural pattern that fits the scale out strategy is *microservices*: many small services where each one of them covers a certain aspect of the application. Individual services can now be scaled on their own, independently from other services. In recent years, it became very popular to run microservices architectures in the cloud, because it simplifies the process of provisioning servers.

To scale well, infrastructure-as-code is an essential factor. This declarative approach allows to define the entire runtime environment in code. Therefore, infrastructure provisioning can be automated. This is necessary since cloud applications often consist of many services running on a large number of machines [Cit+15].

The development of new applications commonly starts as a monolithic architecture where a single service contains the entire business functionality. With this approach, scaling is only possible vertically. In order to allow for horizontal scaling, a transformation of the monolith into a microservices architecture is needed.

1.1 Goal and approach to a solution

This exploratory study should show which approaches are possible to transform an existing application into the cloud. The solution will build on Docker, a technology to perform lightweight container virtualization, enabling a decoupling between application code and infrastructure. The study should give recommendations on how to split up the monolithic architecture. Moreover, it should point out advantages and disadvantages of specific technologies and cloud platforms. Since there is a broad spectrum of existing technologies, the focus will be on technologies provided by Microsoft with the *Azure* cloud platform.

One goal is to automate operations to the greatest possible extent without giving up control of the infrastructure. Additionally, vendor lock-in should be avoided or, if not possible, at least kept to a minimum. Microsoft offers two mature platforms that facilitate the operations of microservice architectures in the Azure cloud: *Azure Kubernetes Services* and *Service Fabric*. Both are platforms that manage the deployment and execution of containers. Important factors when dealing with large complex applications are availability, scaling and networking. Azure Kubernetes Services and Service Fabric are evaluated and compared in detail according to predefined criteria.

1.2 Structure

This thesis is comprised of two major parts, each grouping multiple chapters. Part I explains the transition from a monolith to microservices and is divided into the following chapters:

- Chapter 2 explains the basics of the .NET ecosystem and elaborates the fundamental differences between .NET Framework and .NET Core.
- Chapter 3 discusses the shortcomings of a monolithic architecture and gives reason for a transformation.
- Chapter 4 gives an introduction to the principles of microservices. Furthermore, it evaluates advantages and disadvantages of this architectural pattern.

- Chapter 5 presents an incremental approach on how a monolithic application can be split into microservices.

Part II of this thesis discusses how the microservices created in the first part can be operated in a cloud environment. This part consists of the following chapters:

- Chapter 6 explains the key capabilities an orchestration platform has to implement. Requirements on the orchestration and application level are presented.
- Chapter 7 compares Azure Kubernetes Services and Service Fabric based on the defined key capabilities.
- Chapter 8 reflects on the entire transition to cloud-based microservices and gives a recommendation which orchestration platform to use.

Part I

Transformation

Chapter 2

.NET

This chapter briefly describes the two platforms offered by Microsoft for the development of web applications. Theses platforms are the older *.NET Framework* and the more recent *.NET Core*, which is available since June 2016 [Lan16]. For the development of web applications, Microsoft provides the ASP.NET framework which is available for both platforms. Regardless of the platform, ASP.NET enables the creation of at least two common application types: MVC applications and REST APIs.

2.1 .NET Framework

The .NET Framework is a runtime execution environment for applications and can be used in combination with various programming languages. It consists of two main parts which are the Common Language Runtime (CLR) and the .NET Framework class library that provides reusable functionality to the developer [Micd].

The CLR manages code at execution time and provides core services for managing memory, thread execution, code execution, code safety verification and compilation [Mick]. Part of the CLR is the Common Type System (CTS) which enables the execution of different programming languages on the CLR. The compilation of these languages generates managed code that confirms to the CTS. The generated managed code is then compiled at runtime into native machine language by the CLR, which is called just-in-time compilation. As a result, every language that runs on the CLR is able to use the .NET Framework class library. A widely used language that targets the

CLR and is used for the prototypical implementation in this thesis is C#.

The .NET Framework class library is a collection of reusable types, that can be utilized by the developer. The .NET Framework is only available on Windows operating systems [Mich]. Therefore, it is not suitable for development of cross-platform applications also running on Linux or other operating systems.

ASP.NET is the web framework for the development of websites and web applications in the .NET ecosystem and is based on the .NET Framework. It provides components that facilitate the development of model-view-controller applications and REST APIs. In order to run ASP.NET applications on Windows servers, Microsoft provides a feature-rich web server called Internet Information Services (IIS).

2.2 .NET Core

.NET Core is a development platform and, opposed to the .NET framework, allows the developer to target different operating systems without the need for changes in the application. Currently .NET Core runs on Windows, Linux and macOS and is focused on web applications and cloud workloads [Micg]. Conceptually it is similar to the .NET Framework; .NET Core is built upon its own CLR, which is adapted to the respective operating systems. Furthermore, it is open source and developed under the stewardship of the .NET Foundation. The source code is available on GitHub[1].

In comparison to the .NET Framework, which was a monolithic framework, .NET Core has a modular structure. It is divided into smaller feature-centric packages, leading to shorter release cycles. The NuGet package manager is used to make these packages available.

To deploy an application developed with .NET Core there are two variants: framework-dependent deployment and self-contained deployment. When using framework-dependent deployment, the .NET Core framework is installed on the machine, which hosts the application. All apps deployed to this machine use the same system-wide version. This approach could lead to problems when updating the framework version. The second variant is self-contained deployment where no global framework installation exists. Here the framework is packaged and deployed with the application. This allows updating the framework version for specific applications. The deployed .NET

[1]https://github.com/dotnet/core

Core framework contains the binaries built for the specific platform the application is deployed to. Therefore, the target platform has to be specified when building the application. Because every application is running on its own .NET Core framework, this increases the requirements for disk space when multiple applications are deployed to the same machine [Micf].

Compared with the .NET Framework, .NET Core only contains a subset of the .NET Framework functionality. In particular, only console applications and ASP.NET web applications can be developed. Components for the development of desktop clients, like WPF or WinForms, are not available for .NET Core. Moreover, there is only a subset of the .NET Framework APIs available, but with ongoing development, the number is growing. With .NET Standard, described in Section 2.3, a specification has been created to ease the development of third-party libraries targeting both, .NET Framework and .NET Core.

Since .NET Core is focused on web and cloud applications, ASP.NET has been rewritten based on .NET Core and is now called ASP.NET Core. Whereas the classic ASP.NET, based on the .NET Framework, allows four different programming models to create web applications – MVC, Web API, Web Forms and Web Pages – ASP.NET Core only supports MVC and Web API. ASP.NET Core offers a higher performance than ASP.NET [Micb].

Microsoft provides a cross-platform web server called Kestrel, which is included by default in ASP.NET Core project templates. Configuring the request pipeline is done using the integration of Kestrel in the ASP.NET Core application. This decouples the application from the specific platform. Furthermore, web servers like IIS, nginx or Apache can be used as a reverse proxy for preliminary handling of requests before forwarding them to Kestrel [Mice].

2.3 .NET Standard

.NET Standard is a specification that represents which set of APIs a .NET platform (e.g. .NET Framework or .NET Core) has to implement in order to comply with this specification. Each incremented version number of the .NET Standard specification means that a larger set of APIs is supported.

The specification mostly concerns developers of libraries, which can be used by application developers. If the library targets a specific version of .NET Standard, this library can be used on all platforms that support this specific version. This solves the code sharing problem between platforms. A library developer may want to target the lowest possible .NET Standard version, in order to make the library available for as many platforms as possible [Mici].

Chapter 3

Monolithic Architecture

Before microservices have become popular as an architectural approach to design applications, most web applications were developed in a monolithic way. To decide whether a transformation from a monolithic to a microservices architecture is necessary or profitable, it is essential to know about the downsides and restrictions of a monolith. This chapter discusses the characteristics and limitations of the monolithic architecture that led to the emergence and popularity of microservices.

3.1 Characteristics

A web application following a monolithic architecture is built as a single and self-contained unit. Common enterprise applications consist of three components: A user interface built with HTML and JavaScript, a database which mostly consists of a large number of tables and uses a relational schema, and a server application between the user interface and the database. This server application is the monolith and handles the receiving HTTP requests from clients, generates HTML views that are sent back to the client in response, executes business logic and interacts with the database to query or persists data [LF14].

The monolithic server application is split into modules and services, but everything runs in the same process. Communication between these services is performed directly, either via function calls or method invocations. Therefore, the communication is fast, but has the disadvantage that services are interdependent [LF14].

The deployment of a monolithic application is simple since there is only a single deployable unit, which is the entire application [Ricb].

Horizontal scaling of a monolithic application is only possible by fully replicating it multiple times – this requires the application to be stateless. A server-side load balancer is used to equally distribute HTTP requests to the replicated instances. As shown in Figure 3.1, all instances still access the same database.

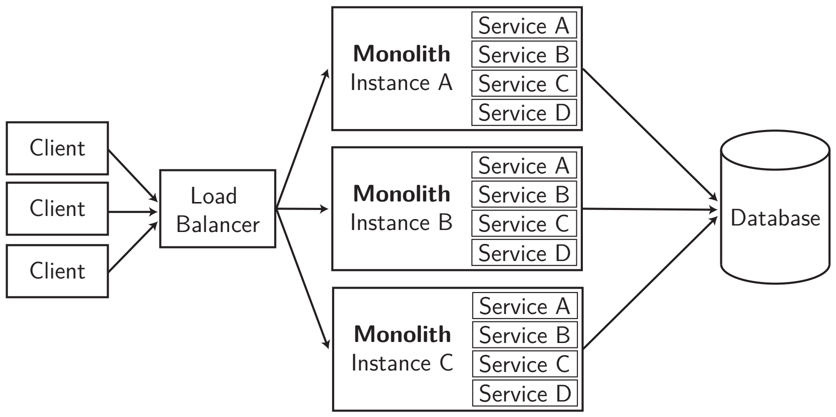


Figure 3.1: Horizontal scaling of a monolithic architecture using three instances of the application

3.2 Shortcomings and reasons for transformation

Although a monolithic architecture is not always bad, it has some characteristics causing disadvantages, especially as the application grows. This section discusses drawbacks resulting from this architectural design.

A common monolithic application is implemented using a single database to persist data. As shown in Figure 3.1, scaling is possible by running multiple instances of the application. As a result, handling a greater load of requests is possible. However, the scalability is not optimal because with each created instance of the monolith, resources for all services are allocated, although only one service is affected by the load of requests [Dra+17].

Within a monolith, changes to the functionality result in a high cost for quality assurance. Due to the high coupling of components, a single modification might affect several features or components throughout the monolith. In addition, manual verification by QA takes place before release when all features are implemented and blocks the release until the whole verification process is completed [GT17].

With a monolithic architecture, it is troublesome and error-prone to follow the continuous deployment approach. Continuous deployment means with each small, incremental change to the source code, a new version of the application is deployed to the production environment. Since there is only a single and self-contained deployable unit, the entire application has to be redeployed. With every redeployment, all running tasks have to be stopped. This creates a large surface for possible errors, and thus monolithic architectures are not well-suited for continuous delivery [Ricb].

As the monolithic application grows in size and complexity, it can get difficult for developers to understand the entire functionality of the application. Especially if a new developer joins the team, it takes time to comprehend the application. This causes the development progress to slow down. Furthermore, it is not possible to have several teams working independently on different business aspects of the application because the services are interdependent and modules are coupled. As a result, changes require coordination between teams [Ricb]. For instance, a slight change to the database schema could affect all teams, requiring them to update their database queries. Dragoni et al. [Dra+17] also discuss the problem of dependencies: Adding or updating them might cause compilation errors and also increase the possibility of unintended application behavior. In 1982 Warner [War82] already identified the monolith's interdependent nature as a problem: Changes to the source code in one part of the application can result in errors in other parts.

Moreover, deployments can again lead to problems if teams do not coordinate them: When two teams are working on two different features, they have diverging code bases. Only one code base can be run on an environment at a time. For example, if several teams intend to test their newly implemented feature, they need to deploy it to the testing environment. Since only one code base can be run concurrently, teams would block each other if they do not coordinate their deployments [Wol16]. Alternatively, each team could have its own testing environment, but this would result in higher resource requirements and greater operational effort.

Following a monolithic approach also has effects on the technology stack. When the entire application is implemented with a certain programming language or framework, it is difficult to adopt new technologies. This can be a problem if the fundamental platform or framework becomes obsolete and will not receive updates anymore [Ric18]. Migrating the application to another platform or framework can be tedious; and since the monolithic application is a single unit, an incremental approach is often not possible.

Chapter 4

Microservices Architecture

The more an application grows in size, the more the shortcomings of a monolithic architecture hinder development. The microservices architecture is a pattern to create web applications in a way that mitigates the disadvantages and limitations of a monolith and enables new benefits that facilitate the operation of an application in the cloud. This chapter covers the characteristics of a microservices architecture focusing on its development. The most important aspects in applying this pattern are explained; in particular, the prerequisites to successfully run a microservices application are addressed.

4.1 Characteristics and principles

Microservices is an architectural term that has no formal definition, but is defined by its characteristics. This section, based on Newman [New15, pp. 1 sqq.], reflects the key concepts of a microservices architecture.

Newman [New15, p. 2] defines microservices as follows: “Microservices are small, autonomous services that work together.” When designing a monolith, related code is grouped together to cope with the application’s complexity, facilitating fast and frictionless development. Modules are used to group the functionality and keep the codebase organized and manageable. Microservices take this modular approach one step further; every module is a separate service running independently of other services. Thus, each single service covers a small part of the application’s functionality [New15, p. 2]. Of great importance is the isolation of each individual service that runs in its own process and uses lightweight communication mechanisms [Fowb].

This isolation enables the key characteristic of a microservices architecture which is the autonomy of each service. Services can be changed and deployed independently of each other without affecting other services. When changing a service, consuming services must not be required to adapt to this change, as this would mean a tight coupling between these services. To accomplish this autonomy, services hide their internal implementations and only expose a defined API to be consumed by other services. If service A needs to communicate with service B, it may only use the defined API of service B. Since service A does only know about the defined API of service B, the internal implementation of service B can be changed and redeployed without affecting service A [New15, p. 3]. In order to keep service internals hidden, each service has its own data storage. Services are not allowed to directly access the data storage owned by another service because this would result in a coupling between the accessing service and the owning service's domain model. To acquire data owned by another service, the exposed API has to be used [New15, pp. 39 sqq.]. In Section 4.4 inter-service communication is covered in more detail.

Because each service runs in its own process, communication is done only via network calls. This points out a disadvantage of a microservices architecture because network calls are more expensive than in-process calls. To compensate this, service APIs are designed more coarse-grained and aggregate data from multiple in-process calls in one network call [Fowa]. Furthermore, refactorings involving multiple services are more difficult: Moving functionality from one service to another requires more work than moving only inside a process boundary, as it most likely requires changes to the exposed APIs. A coarser-grained API makes this even more difficult [LF14].

Considering the size of a microservice, which the term itself emphasizes, there is no clear answer. Wolff [Wol16] gives a number of guidelines on how to find the right size for a microservice and, instead of specifying concrete measures, defines upper and lower limits: A single microservice is developed by one team, and therefore it may not grow to a size, where multiple teams are needed for development. Since microservices are used to modularize an application, they must be so small that a developer can still understand the entire functionality of the service. Additionally, a microservice should be replaceable: An increasingly complex and expensive maintainability might be a reason to replace a microservice. Also, switching to a more powerful technology could bring significant improvements and requires the service to be replaceable. Replaceability sets an upper limit for the service size. Factors that influence the lower limit are the additional cost to provide infrastructure for a service, the communication overhead due to network calls and the required data consistence which can only be guaranteed within an individual service [Wol16, p. 33].

4.2 Modular design

As explained above, characteristic for the microservices architecture is its modular design, where all services work independently of each other. This section explains what to consider when splitting an application into smaller, independent parts.

4.2.1 Loose coupling and high cohesion

Two key concepts, which are very much applied in object-oriented systems, also help to divide an application into several independent services: loose coupling and high cohesion. Loose coupling means that each microservice has to be able to be changed and redeployed without interfering with other services. The opposite – known as tight coupling – would be if changes to a service's internal implementation would require other parts of the application to be adapted as well. As a result, this service cannot be deployed independently. To accomplish loose coupling, services may only know as little as possible about other services. Furthermore, it is also important to reduce the communication between services to a minimum [New15, p. 30].

In order to facilitate changes to a service, related behavior is grouped together. Thus, updating an application only requires a change in one place. This concept is called high cohesion and allows to release updates quickly. Without high cohesion, an update to the application is likely to require changes to several services, which then have to be released simultaneously. This is slower and increases the risk of unsuccessful deployment [New15, p. 30]. The concept of high cohesion adheres to the single responsibility principle defined by Martin [Mar10]: "Gather together those things that change for the same reason, and separate those things that change for different reasons."

4.2.2 Bounded context

When it comes to the question how to split an application into several microservices, Newman [New15] and Wolff [Wol16] both refer to the bounded context pattern defined by Evans [Eva03]: Representing the domain of a large application in form of a unified model is difficult. It is highly possible that different teams use slightly different representations of the exact same thing they are modeling. Multiple coexisting models impede clear communication between developers. This can be solved by dividing the application domain into several bounded contexts. A bounded context defines the part of the domain where a specific model is applicable. Within this context, the model acts as a ubiquitous language and simplifies the communication. The context is framed by an explicit boundary. Everything outside of this boundary does not concern the context or its model. At the points across the boundaries where different bounded contexts interact, a translation of the model is necessary [Eva03, pp. 335 sqq.].

For an exemplary domain of a webshop application, two separate bounded contexts could be the product catalog and the warehouse. Both have different models of a purchasable product. The ordering system holds information about color, size or description, whereas the warehouse system holds information about the stock.

The defined boundaries provide a blueprint for the division of the application into microservices. Hereby every bounded context is represented by an individual microservice. It is also possible that a single bounded context is divided into multiple microservices. For instance, this would be reasonable if one part of the bounded context has to be scaled out more than others. However, a single microservice should not cover multiple bounded contexts [Wol16, p. 45].

4.3 Data persistence

It is of particular importance that each service has to own its private data storage and must not share it directly with other services. While it is common for monolithic architectures to have a single database, microservices use a decentralized approach for data persistence. Each service can decide individually which data persistence technology is the most suitable for its needs. For example, this could be a relational database or any kind of NoSQL database like a graph database or a key-value database. Although it is also possible for a monolith to use different kinds of databases, it is a salient characteristic of a microservices architecture to combine several storage technologies to optimize data persistence – this is called polyglot persistence [LF14].

As explained above, when designing microservices according to the bounded context pattern, their contexts share boundaries. In comparison, in a monolith these contexts would be linked by a simple foreign key relationship within a single database. Whereas in a microservices architecture, there is also a boundary of different databases. Therefore, a mechanism is needed

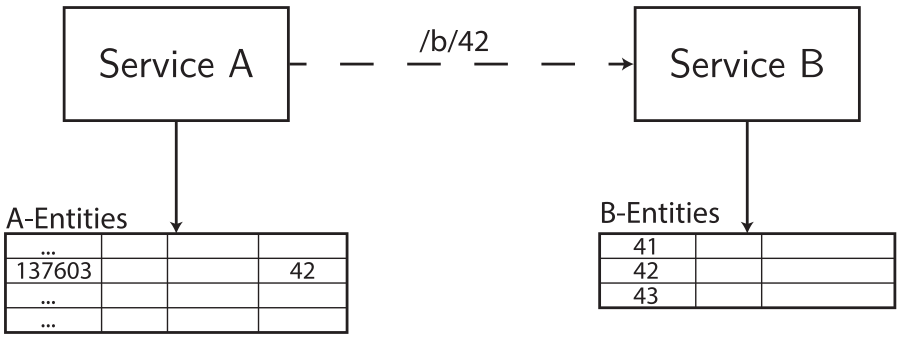


Figure 4.1: Service A accessing an entity with id 42 owned by Service B; adapted from [New15, p. 85]

to represent the foreign key relationship. A microservice is not allowed to access a data storage owned by another microservice because this would introduce a tight coupling between these two services: If the owning service updates its data model, the dependent service also has to update its model. Both services have to be deployed simultaneously; otherwise, the data access would break. To avoid this, the owning service exposes an API that provides the data to other services. Figure 4.1 illustrates this approach. As described in Subsection 4.4.1, this can also be performed in reverse direction by propagating the data from the owning service to dependent services in an asynchronous manner. On no account, dependent services are allowed to directly access databases they don't own [New15, pp. 82 sqq.].

4.4 Integration and communication

Microservices should be decoupled in order to allow independent development of each service. In addition, the implementation of a microservice should not be restricted to any given technology. Therefore, communication between microservices should be technology agnostic. This allows services to consume other services without having to deal with technological hurdles and without being forced to use a particular technology [New15, p. 40].

4.4.1 Types of communication

The most common approaches used for service communication are remote procedure calls (RPCs) and lightweight messaging. RPCs follow a simple, synchronous request-response pattern, e.g. HTTP – often in form of RESTful APIs. A benefit of HTTP is that it has built-in caching, which can be leveraged to reduce the number of requests for seldom changing data. In contrast to synchronous RPCs, messaging is an asynchronous pattern that decouples services even more. The messaging infrastructure does not contain business logic and is only responsible for routing the messages. The entire business logic resides in the distributed services [LF14]. Of particular importance is that a lightweight message bus is used because the cost of an enterprise service bus (ESB) and its associated connectors is high [GT17].

Synchronous and asynchronous communication

To integrate microservices, a variation of communication styles can be used. These styles can be classified by different characteristics. The characteristic that affects the application design the most, is whether the communication is performed synchronously or asynchronously.

Synchronous communication means that a caller makes a call to a service and waits until this service returns a response; i.e. the caller blocks until the operation is completed. This is similar to the usual in-process communication in the form of method calls. In comparison, with asynchronous communication, the caller does not wait for a response from the called service. Thus, the caller cannot be sure if the called service successfully processed the call; most often the caller is not even concerned by the result. In case the caller does need the result, it could receive it via a succeeding asynchronous call in reversed direction. This approach can be useful for long-running calculations where it would be too expensive to keep open a connection between caller and callee. Asynchronous communication increases the decoupling of services and helps to build a better scalable application but also increases the complexity of the overall application. Especially for long-running calculations if the result will be further processed by the caller: It could be possible that the caller has to keep state for further processing or that this particular instance of the caller has crashed or has been removed from the application in the process of scaling down [New15, pp. 42, 57 sq.].

Torre et al. [TWR17] recommend to keep internal communication between microservices at a very minimum but when integration is needed, it should be done entirely in an asynchronous manner. That being said, a microservice still provides an API which can be called synchronously by clients. Such clients could be a single-page application, a mobile application or an MVC service. However, the successful processing of a client request by a certain microservice may not depend on further synchronous calls to other microservices during this processing. A microservice depending on synchronous calls has a negative impact on the overall response time. Moreover, services are not totally autonomous anymore and a failing service in this synchronous call chain negatively impacts the application's resilience. In case a service needs data owned by another service, it is advised to not rely on synchronous queries. But in order to have this data available, it should be replicated into the dependent service's data storage. This data can likewise be provided by asynchronous messages [TWR17, p. 46]. Following this approach results in eventual consistency of the data; this term will be explained in Section 4.5.

Commands, queries and events

Besides the synchronicity, communication patterns can also be classified in three types of collaboration: queries, commands and events. A collaboration between two services can combine multiple styles [Hor17, pp. 79 sqq.].

Queries are used when a service needs information from another service. A query is done in form of a request-response pattern, i.e. synchronously. As mentioned above, synchronous communication between internal microservices should be avoided; therefore, queries are only applied by clients outside the application to request data from a microservice. When using HTTP, a query would conform to a GET request [Hor17, pp. 80-82].

Commands are used when a service needs to have another service perform an action without needing a response. Commands can be sent synchronously [Hor17, pp. 82-84], conforming to a HTTP POST or GET request, or also asynchronously in form of single-receiver message-based communication [TWR17, pp. 50 sq.].

Events are used when one or more services need to be informed about changes from another service. This type of collaboration is asynchronous because the service, which has updated its data, does not call the subscribed services directly. Since there is no direct call, subscribed services may not receive or process the event immediately. There are two ways how event-based collaboration can be implemented: The first one is to use a message-based approach. All services that want to receive information about a particular event, subscribe to a central message bus. When this event occurs, the service publishes it to the message bus which then routes the event message to the subscribed services. The subscribed services do not need to know anything about the publishing service; they only subscribe to a certain topic, to which the events are published [TWR17, pp. 51 sq.]. The second approach is to have services expose an event feed that is polled by subscribed ser-

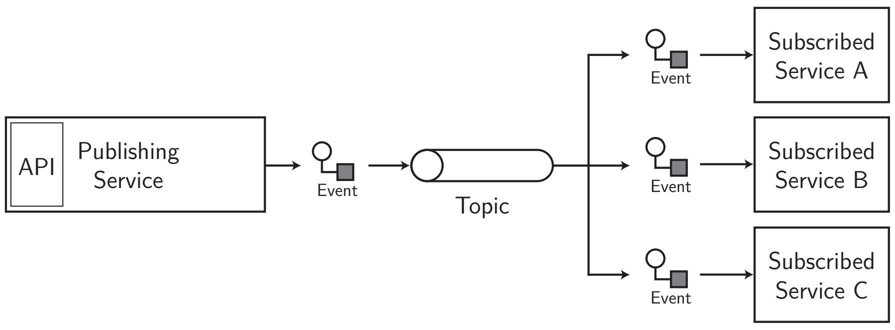


Figure 4.2: Event-based collaboration using a message bus

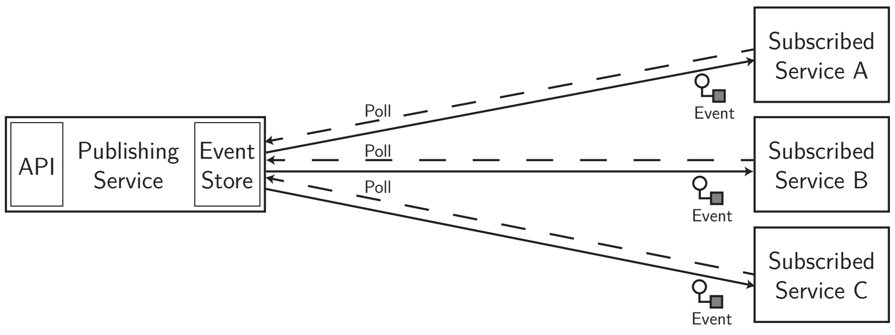


Figure 4.3: Event-based collaboration using an event store

vices using RPCs. For example, the event feed could be reachable via an HTTP endpoint. Subscribed services send GET requests to the endpoint and receive new events, which they can process subsequently. Although the RPC-based polling itself is synchronous, the collaboration is asynchronous because pushing the events into the event store and polling them is not linked temporally [Hor17, pp. 85-87]. Figures 4.2 and 4.3 show both approaches in comparison.

4.4.2 Fault tolerance

A microservices-based application is a distributed system; this implicates that failures will occur eventually. Many of these system failures cannot be known at design or build time; and, thus the running system has to be able to cope with them [Vog16]. When designing a microservices architecture, it has to be taken into account that services will fail or become unreachable. In this cases of failure, the rest of the application still has to function without the failed services. For example, in case the basket service fails in a webshop application, users are still able to view the product catalog. To handle and mitigate failures, several stability patterns can be applied [Nyg07, pp. 110 sqq.], which are explained as follows.

Timeouts and retries

For outbound requests, timeouts are applied to fail fast if it's unlikely that a call to a service will return successfully and it is reasonable to stop waiting for the call to return. Timeouts can be combined with a retry pattern. Here a failed call will be re-executed. The problem with immediately re-executed calls is that they are highly likely to fail again. From the perspective of an application user it would be preferable to fail fast and return a failure result. A sensible default timeout would be about 250 milliseconds, of course, depending on the individual use case [Nyg07, pp. 111 sqq.].

Circuit breakers

The circuit breaker pattern is used to prevent the execution of a call if it is known to have failed before. This allows to fail fast because no timeout has to be awaited. All calls to a remote service go through a circuit breaker component that keeps track of the service's health status [Nyg07, pp. 115 sqq.].

The circuit breaker has three states, as shown in Figure 4.4, which signal the health of the service to be called. A circuit breaker's initial state is closed. All calls to the remote service are passing through. In case a call fails, the circuit breaker increases a failure count. When this count exceeds a defined threshold, the circuit breaker goes into the open state. In this state, no calls are passed through to the remote service, but immediately throw an error. After some time, when it's possible that calls will succeed again, the circuit breaker goes into a half-open state and the next call is passed through to the remote service. If the call succeeds, the circuit breaker switches into the initial closed state. If the call fails, it switches into the open state. It is possible to distinguish between failure types (e.g. timeouts or various HTTP error status codes) and apply different thresholds [Nyg07, pp. 115 sqq.].

Another approach to go from open to closed state is to use health checks instead of going into the half-open state. The circuit breaker sends health check calls to the remote service and, depending on the response, resets into

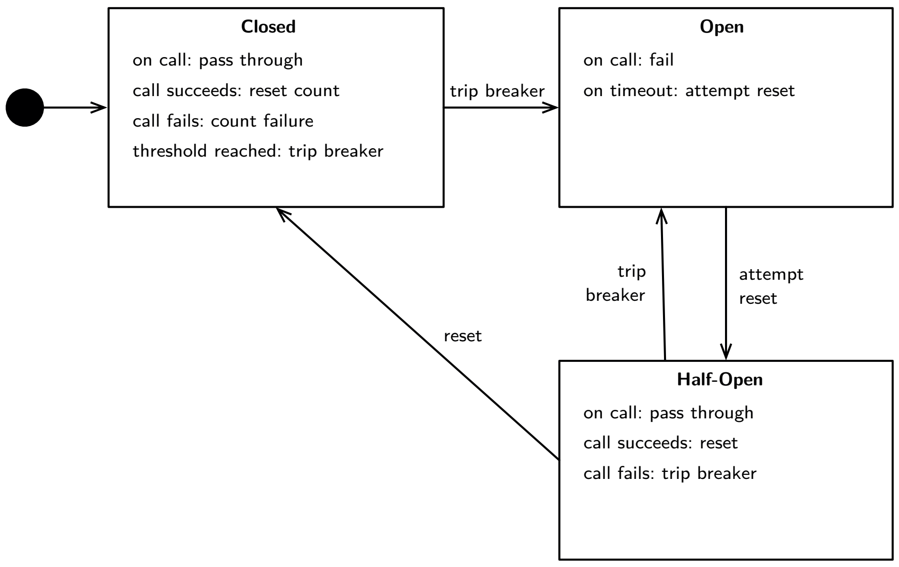


Figure 4.4: States of a circuit breaker [Nyg07, p. 116]

the initial closed state [New15, pp. 212 sq.].

It is important to log state changes of a circuit breaker as it can be used as a metric to indicate problems in the application [Wol16, p. 204].

Bulkheads

Bulkheads are a pattern applied to limit the damage of an entire system if a part of it fails. The term comes from the metal partitions that are used to divide a ship into watertight parts; if a leak occurs in the ship, the bulkhead prevents water from moving into all parts – i.e. only one part is damaged but the rest of the ship stays intact [Mic17a].

When a consumer sends requests to multiple microservices in parallel, it allocates resources for each request. If one of these services does not respond, the consumer's connection pool will become exhausted since all calls to the unresponsive service block resources. As a result, the consumer is no longer able to send requests to healthy microservices because it can no longer acquire any more connections. By applying the bulkhead pattern, separate connection pools for each service are used. Thus, requests to healthy services are still possible if one service is unavailable [Mic17a].

Caching

A caching mechanism can be used to make an application more fault tolerant. By implementing client-side caching, it is possible to serve data to the user, even if a microservice is not available. Since HTTP has a built-in caching concept, the implementation of caching is simplified. Although the cached data may be outdated, serving stale data to the user is often preferable to returning an error. Besides client-side caching, the same resilience could be achieved using a reverse proxy, where the data is served by a proxy server between server and client. This may be preferable to client-side caching due to its simple integration [New15, pp. 225-228].

4.4.3 Versioning

Reasons to build applications in form of a microservices architecture is to develop each service independently. This individual, rapid evolution of a service may impact consuming clients, which rely on the service's API. Requirements for a service will eventually change and so will its exposed API; but changes to the API must not break consumers.

Semantic versioning

Versioning the API is an approach to communicate changes to consumers. Semantic versioning is a specification that describes how version numbers should be applied: The specification defines a version schema consisting of the form `MAJOR.MINOR.PATCH`. The major version is incremented when introducing incompatible API changes, which would break consumers of the previous major version. Incrementing the minor version signals added functionality in a backwards-compatible manner. The patch version is incremented for bug fixes, which are also backwards-compatible [Pre].

When releasing a new major version, the old major version still has to remain available for some time in order not to influence consuming clients and give them time to adapt to the new major version. This can either be done by exposing two different endpoints in one service or by running two services with different versions concurrently [New15, pp. 64-67].

Tolerant reader

The tolerant reader pattern is about the consumer expecting changes to the API. For example, when a consumer calls an API, it only uses the fields of the response it needs and ignores the rest. Therefore, if a response of a modified API returns additional fields, these do not impact the consumer [Fow11]. The tolerant reader pattern adheres to the robustness principle, also known as Postel's law, which states: "Be conservative in what you do, be liberal in what you accept from others [Pos80]." An anti-pattern is to generate classes on the consumer side from a given schema provided by the API. Updating the API often breaks the consumer [Fow11].

Consumer-driven contracts

Another pattern to keep the number of breaking changes to a minimum is called consumer-driven contracts. These contracts express expectations of the consumer to the API provider and show how a consumer uses the API. Consumer-driven contracts are represented in form of tests that assert the expectation to the provider. These tests are available to the provider and give insights how the consumers implement the API. As a result, the provider can determine if a change would introduce failing tests and consequently break a consumer. Furthermore, consumer-driven contracts also reveal if parts of an API are not used at all. This could help the provider to refactor and optimize its API. A limitation of this pattern is that it only works if the consumers are known as its the consumers responsibility to provide the contracts. Therefore, this pattern cannot be applied to public APIs [Rob06].

4.4.4 Exposing services to external clients

In a microservices application only certain services should be exposed to external clients. The API gateway pattern acts as a facade for the internal microservices. It provides a single point of entry for all external clients and routes the requests to the back end microservices. Additionally, an API gateway can also take over responsibilities from the microservice, for example terminating incoming HTTPS connections or authorizing the client [Rica].

An API gateway can improve performance by aggregating requests to multiple microservices. Particularly for mobile devices, this can reduce the required amount of round trips between client and server. Furthermore, the API gateway can strip information that is not needed by a particular client, resulting in a smaller request payload [Sti15].

A specific approach to implement an API gateway is the backends for frontends (BFF) pattern. Here, for each type of client a specifically adapted API gateway is implemented [Rica].

4.5 Eventual Consistency

The CAP theorem by Brewer [Bre00] states that a distributed system, consisting of multiple nodes or instances, can only fulfill two of the following three guarantees at any given time:

- **Consistency**: All nodes in the system return identical results at all times and the results are the most recent state.
- **Availability**: Every node always returns a result within an acceptable time.
- **Partition tolerance**: The system is still operational even if a part of the nodes fail.

As already mentioned, a characteristic of a microservices architecture is that the overall application is still functional to a certain degree if individual services fail. Therefore, a microservices architecture has to comply with partition tolerance. Depending on the requirements to an application, it either has to sacrifice availability or consistency. A common reason to use microservices is the need for a highly scalable application. If the consistency requirements allow that data is not the most recent, an AP system fits the needs best.

A system with weak consistency does not guarantee that an update to a data object will be visible to subsequent accesses. In the context of distributed storage systems, Vogels [Vog09] defines eventual consistency as a special form of weak consistency. This means the system guarantees that subsequent accesses will eventually return the most recent value if no further updates are made to the data object.

This definition can also be applied to microservices: When data is sent between services using asynchronous messaging, as covered in Subsection 4.4.1, message-receiving services may not have the most recent data, because of the delay in message transmission. The application is eventual consistent because messages are received eventually and the data can be updated.

Compensating transaction

Business processes often span over multiple microservices, e.g. an order in a webshop application could involve a basket service, an ordering service, a warehouse service and a payment service. During a checkout, the application might become inconsistent and after the process has been completed successfully, the application becomes consistent again. The problem with eventual consistency is when a step in the business process fails and some already completed steps need to be undone. This is accomplished using a compensating transaction. A simple reset of the state might not be possible if other changes have already been applied to the data. Therefore an intelligent workflow is needed for the business process, defining how successful steps can be undone [Mic17b].

Since a compensating transaction is also eventually consistent, it can fail, too. A solution to this problem is to implement compensating transactions as idempotent commands. Therefore, failed compensating transactions can be retried [Mic17b].

As a last resort to fix the inconsistent state of the application, either an automated background process can be used [New15, pp. 91 sq.] or manual intervention can be performed [Mic17b].

4.6 Advantages and disadvantages

Monolithic architectures have a number of shortcomings, as covered in Section 3.2, that restrict an applications growth once it has reached a certain size. As a result, the time for development of new features increases significantly. Microservices architectures have evolved for the reason of mitigating the monolith's downsides and to overcome its limitations. But this architectural pattern also comes with new challenges. This section emphasizes the benefits and drawbacks of microservices, focusing on the technical perspective.

4.6.1 Advantages

The main characteristic of a microservices architecture is its distribution over independently functioning services. This brings the following benefits.

Scaling

The modularity of the application architecture and the independence of services allows to horizontally scale individual services. Several instances of a microservice can be run in parallel. The number of instances can be scaled independently of other microservices; this enables a precise load balancing. In comparison, load balancing in a monolithic architecture is accomplished by scaling instances of the entire application [New15, pp. 5 sq.].

Since the load on a web application varies over time, creating and removing service instances allows the application to adapt to the load. Furthermore, horizontal scaling enables better utilization of resources and, in combination with on-demand provisioning of cloud resources, results in a more effective cost control [New15, pp. 5 sq.].

Resilience

Due to the service independence, failures are isolated and do not cascade through the entire application. A correctly designed architecture is important to cope with service failures. Since failures will occur eventually, the application may degrade in functionality but still remains in a state where it is able to successfully serve a part of the requests. Owing to the modular design, a microservices application can be distributed over different physical servers, and therefore mitigate the risk of hardware failure. The distributed nature of microservices requires them to communicate via a network. But networks aren't reliable and introduce a new possible source of failure which has to be considered when designing an application [New15, p. 5].

Deployment

Deploying a monolith brings a high risk because the application has to be deployed as a whole. As a result, deployments are done infrequently and it takes time until new features can be release. In a microservices architecture, each service can be deployed and released independently which enables shorter release cycles. Also, problems with a new release can be identified and isolated quickly. Short release cycles require a high degree of automation. Approaches like continuous integration and continuous delivery allow to run every commit against several test suits and then integrate the changes into production using a pipeline system [New15, pp. 103 sqq.].

Heterogeneous technology

A microservices architecture allows to use a different technology for each service implementation. In general, developers are free to choose the language, frameworks and databases to design a microservice that fits the requirements. Moreover, it is also possible to implement the first prototype in any new language and then, depending on the gained experience and learnings, continue the development or restart with a different technology stack [New15, pp. 4 sq.]. In agile environments, requirements are likely to change and the development needs to be adaptable [WC03]. With the same approach, an entire microservice can be rewritten with a better suited technology stack [New15, pp. 4 sq.].

All microservices share common functionality that every service must implement, e.g. logging, handling of failures in other systems, subscribing and publishing to the message bus, health checks, etc. To reduce this initial effort, service templates which implement all common functionality for a specific technology can be utilized. By leveraging these templates, a team can immediately get going with the implementation of business functionality. The effort of providing such service templates is only reasonable if the allowed technology stack is restricted. This is contradictory to a completely heterogeneous technology [New15, pp. 22 sq.].

Team structure

When developing a monolith, teams are often organized according to their responsibility, e.g. a team of front end developers, a team of back end developers, a team of database administrators, etc. Team sizes increase with the growing application. Applying the microservices pattern makes it possible to break them down into cross-functional teams organized around services. This reduces the team sizes, and thus enables the teams to be more productive. Each team is responsible for the development of its microservice. When implementing features within a microservice, unnecessary coordination overhead between multiple teams is eliminated. Only changes to service boundaries require coordination with the affected teams [New15, pp. 7, 191 sqq.].

According to Conway's law, an organization will produce an application design that is a copy of the organization's communication structures [Con68]. For the correct utilization of the microservices pattern, it is therefore essential that the teams are properly structured and aligned with the service boundaries. Splitting an organization into several loosely coupled teams on purpose in order to achieve loosely coupled services is called the Inverse Conway Maneuver [Tho].

To handle the coordination overhead, a core team can be formed. Members of each service's team are represented in the core team. This team is responsible for architectural decisions and coordinates refactorings that require to move functionality from one service to another [BHJ16].

Rapid development

In a microservices architecture, each team owns its services and is solely responsible for its development. If a team has to require permission in order to integrate the API of another service, this blocks and slows down the development. By eliminating the requirement for approval, a high degree of innovation can be realized [Kil16].

Within a monolith, the deprecation of functionality is difficult; whereas within a microservices architecture, calls to a service's public API can be analyzed to find out which other microservices still use the service. If another team still depends on the deprecated service, the service ownership is transferred to this team and it is now responsible for the maintenance. Due to the limited human resources a team has, it is required to migrate or terminate the dependency to the deprecated service. Then the deprecated service can be eliminated [Kil16].

4.6.2 Disadvantages

The distribution of services strongly facilitates fast development of large applications, but also entails some challenges that one has to be aware of.

Latency

Microservices are distributed and communicate over a network. This implicates latency and a longer response time. Depending how the microservices are distributed – in a single data center at one location or multiple data centers across the globe – this latency can be fairly high. This problem can be alleviated by reducing the need for communication between microservices. For example, different approaches are building a more course-grained

API or providing batch endpoints to combine several requests. But after all, communication between microservices cannot be eliminated completely. Another approach to reducing latency is to run services in the same data center or even on the same physical machine if they share a boundary and need to communicate [Wol16, pp. 69 sqq.].

Identifying boundaries

In a microservices architecture, refactoring that only concerns a single service is simple. If the refactoring involves multiple services and functionality has to be moved across service boundaries, it gets complex. Due to possible different technology stacks, code often cannot easily be moved. Functionality might be rewritten with a different framework or even with a different language [Wol16, pp. 74 sqq.].

At the start of a new project, requirements and business domain often are not completely clear or, depending on the agile nature of many projects, requirements will change over time. Designing the right architecture and splitting the application domain into correctly sized microservices is a difficult task and often leads to problems at the first attempt [Wol16, pp. 74 sqq.].

Overall complexity

A microservices architecture introduces a greater overall complexity. Microservices expose APIs to be consumed by other services. These APIs are contracts and it requires effort to evolve them without breaking consumers. Concepts like versioning or consumer-driven contracts can help with the API design, but do not eliminate the complexity entirely [New15, pp. 62 sq.].

As discussed in Section 4.5, strong consistency in distributed systems often restricts the scalability, and therefore an eventual consistency model is applied. This results in new challenges where the application has to cope with stale data and must be able to recover from data inconsistencies. This often involves the underlying business process that spans over multiple microservices and requires coordination between development teams [Wol16, p. 73].

Since communication is not reliable and failures will eventually happen, microservices require a considerable amount of work to make them fault-tolerant. Furthermore, to ensure an application's resilience, monitoring of each individual microservice is important in order to detect and analyze errors. This introduces the need for a monitoring system that gathers all metrics [Wol16, p. 77].

Dividing an architecture into separate, independent services exposes many boundaries which are a possible source of error. To ensure correct behavior across these boundaries, additional testing is needed. Clemson [Cle14] discusses several kinds of testing strategies, which range from fine-grained to coarse-grained focus: unit tests, integration tests, component tests, contract tests and end-to-end tests. The more coarse-grained focused they are, the more services are involved and the more effort the tests require writing.

Chapter 5

Splitting a Monolithic Application

This chapter shows how an application that was built following a monolithic architecture, can be transformed into a microservices architecture. As a read-world example serves an application named *allReady*, which is available as open source. In a prototypical approach, this application is split into microservices. The bounded context pattern is applied and evaluated on its practicability. Goal of the transformation process is to obtain an application architecture consisting of independent services, which can be deployed and scaled individually, in order to leverage the benefits of microservices. As a method to split the monolithic architecture, an incremental approach, which extracts one service after another, is applied and evaluated. In the end of this chapter, the overall transformation process is discussed; in particular, difficulties and hindrances that occurred during this process are addressed.

5.1 allReady – an open source monolith

The open source software allReady[1] serves as an example of a monolithic application. It is a platform for preparedness campaigns delivered by humanitarian and disaster response organizations. The main focus is on decreasing the impact of disasters. For example, these disasters could be storm floods, where people need to find a place to sleep; earthquakes, where emergency responders need to be coordinated; or droughts, where food supplies need to be managed. Using allReady, local communities can prepare for disasters by coordinating volunteers with different skill sets. Furthermore, it makes it

[1] https://github.com/HTBox/allReady

easier for volunteers to get involved and lowers the barrier to start participating.

Development of the application has started in July 2015 and since then over 3600 commits from over 160 contributors have been made.

5.2 Solution architecture

allReady gives the user two options to interact with the application. They can either use the website available in the browser or access the platform with the mobile app on a smartphone. Both clients are provided with data by a monolithic application that is developed using .NET Core 2.0. This monolith provides REST endpoints for the mobile app and also renders the HTML views for the web application. The monolithic application is designed following the MVC pattern. Figure 5.1 shows the architectural design.

Besides the domain-specific logic, the monolith also manages user authentication. For this purpose ASP.NET Identity[2] is used. Once a user has logged into the web application, a cookie is set to authenticate further requests. For the mobile app a custom implementation of authentication tokens is used: The application assigns a personal token to the user. This token is sent with every request and is validated by a middleware implementation on the server side before executing controller logic. The library Hangfire[3] is used for the execution of recurring jobs.

For sending email and text message notifications a service is implemented with Azure WebJobs[4] targeting .NET Framework. When the monolithic service pushes messages into the queue, the notification service is triggered and processes the messages. Depending on the message, it sends HTTP requests to third-party services which then send emails or text messages.

5.3 An incremental approach for splitting the monolith

To transform the monolithic architecture of allReady into microservices, an incremental approach is taken. With each increment, one service is extracted

[2] https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.0

[3] https://www.hangfire.io/

[4] https://docs.microsoft.com/en-us/azure/app-service/web-sites-create-web-jobs

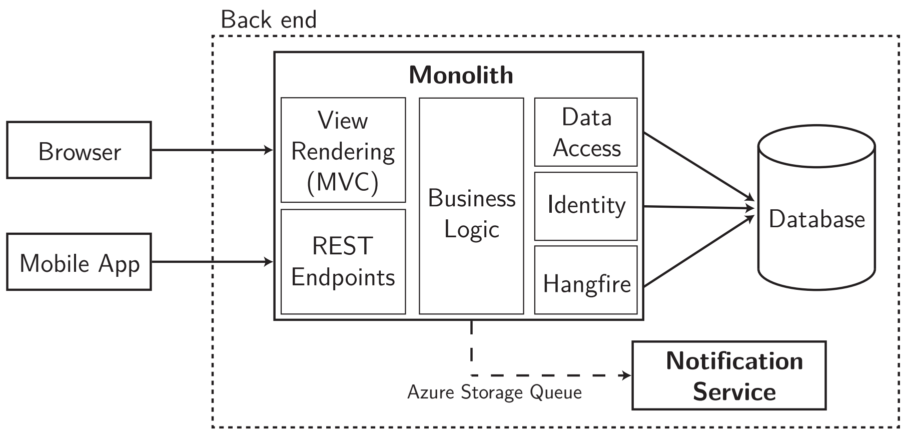


Figure 5.1: Monolithic architecture of allReady

from the monolith to achieve better decoupling and a higher cohesion of business capabilities.

5.3.1 Identity service

As a first step, the bounded context pattern, explained in Subsection 4.2.2, is applied to identify the boundaries on the basis of which the application can be divided into separate microservices.

The monolithic application utilizes ASP.NET Identity which uses several entities to model the access control for the application users. The user entity consists of multiple fields storing personal information and also data about the authentication process. These entities are persisted in the same database as the main model which holds information about allReady's business capabilities. The main model references the user entity to model relationships, e.g. a campaign is managed by an organizer that is represented in form of the user entity. The analysis shows that most user entity fields required in the identity model are unused when performing business logic. In fact, only the user ID is required. Therefore, the identity model can be separated into an individual context. In order to interact across the context boundaries, a user entity is existing in both contexts. Figure 5.2 shows the identified contexts.

To extract the identity context into a separate service, the framework IdentityServer4[5] is used. It is available for .NET Core 2 and provides additional

[5]http://docs.identityserver.io/en/release/

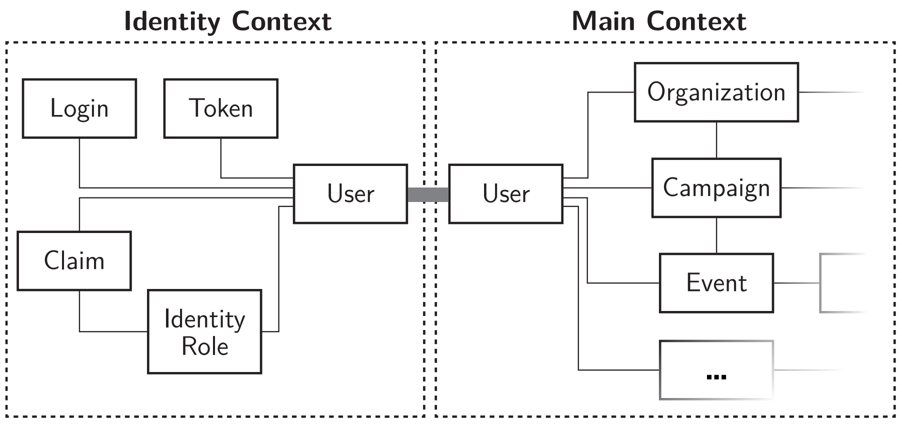


Figure 5.2: A simplified depiction how the application domain is separated into two bounded contexts

packages to utilize ASP.NET Identity as the underlying data abstraction. This enables a faster service extraction because no database schema has to be adapted or rewritten; thus, the identity schema is moved from the monolithic database into a new one that is now owned by the identity service.

Many existing database queries in the main database rely on the referential integrity to the user entity. In order to not break these queries, the user entity is not removed from the main schema entirely but only the required fields are kept – for the prototypical implementation this is only the ID. Further fields might be replicated from the identity service.

In case a new user has registered with the identity service, they are still not available in the main service. For instance, they cannot be assigned to an event as a volunteer. To propagate new users, the identity service publishes events to which the main service is subscribed. Since the processing of the received event in the main service is eventually consistent, it might take some time before the user is replicated to the main database and can be assigned to an event.

To enable publishing and subscribing to events, a message bus is introduced into the application architecture. The open source software RabbitMQ[6] is utilized, which provides the needed functionality for sending messages. Because a dependency to a specific message bus would introduce unnecessary constraints, all microservices only use an abstraction of the message bus.

[6]https://www.rabbitmq.com

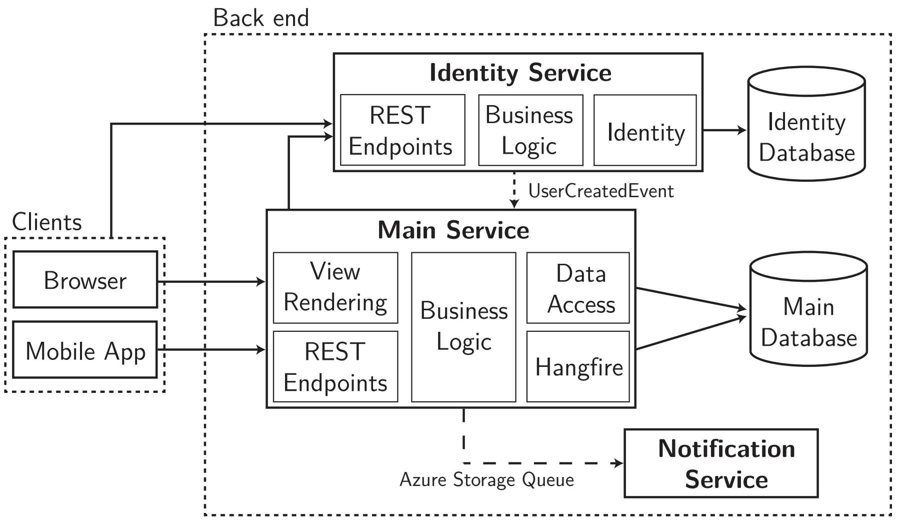


Figure 5.3: Extraction of the identity service

For several views the main service needs to display information of multiple users existing in the application, e.g. a list of all volunteers for an event. To populate this view with the volunteers' names, the view rendering component queries the identity service. The identity services provides a REST endpoint to serve the user information. In order to protect this endpoint OAuth 2.0 and OpenID Connect are used.

Authorization and authentication

IdentityServer4 implements the OAuth 2.0 and OpenID Connect protocols and enables an authentication and authorization workflow for different clients. OAuth 2.0 is a protocol to enable authorization in services and uses an access token that is validated. The validation process checks if the access token is granted the required scopes to access an endpoint. OpenID Connect is a layer built on top of OAuth 2.0 and enables authentication by providing information about the user's identity. This information is embedded in so-called claims that are embedded in an id token.

By using IdentityServer4, both clients – web application and mobile app – are able to use the same protocol instead of using a cookie based authentication for the web application and a custom token-based implementation for the mobile app. Figure 5.3 shows the application architecture in a simplified form after extracting the identity service.

```
 1 public static IEnumerable<IdentityResource> GetIdentityResources() {
 2   return new List<IdentityResource> {
 3     new IdentityResources.OpenId(),
 4     new IdentityResources.Profile(),
 5     new IdentityResources.Email(),
 6     new IdentityResources.Phone(),
 7     new IdentityResource(
 8       name: "ar.profile",
 9       displayName: "AllReady Profile",
10       claimTypes: new[] {
11         AllReadyClaimTypes.Organization,
12         AllReadyClaimTypes.UserType,
13         AllReadyClaimTypes.TimeZoneId,
14         ...
15       })};}
```

Program 5.1: Definition of identity resources within the identity service; lines 3 to 6 define standard claims from the OpenID Connect protocol [SBJ14], whereas lines 7 to 15 define custom claims that are required by the main service

The identity service's purpose is to issue identity and access tokens which are then used to authenticate and authorize the communication between client and services. Within the identity service, resources are defined that enable authentication and authorization. Here a distinction is made between identity resources for authentication and API resources for authorization. Identity resources represent claims of an identity, e.g. the email address. API resources represent endpoints and are used to restrict client access, e.g. a specific client might only have access to certain endpoints.

Program 5.1 shows the configuration of identity resources that will be embedded as claims in an identity token and are used for authentication. Besides standard claims from OpenID Connect, the `ar:profile` claims are used to add further information concerning allReady to the user identity. In Program 5.2 is shown how API resources are defined. Currently only a resource for the users endpoint, which is located within the identity service, is specified. When additional microservices are added to the application, their exposed APIs need to be added as an API resource in order to manage authorization of different clients.

To access the specified identity and API resources, each client has to be configured within the identity service. Of particular importance are the allowed scopes that define which resources a certain client is allowed to access. Program 5.3 shows the definition of the main service called *MVC client*. This client has access to all existing identity and API resources.

```
public static IEnumerable<ApiResource> GetApiResources() {
  return new List<ApiResource> {
    new ApiResource("users", "User API within auth service")
    // add further microservice APIs here
    };}
```

Program 5.2: Definition of API resources within the identity service

```
 1 public static IEnumerable<Client> GetClients() {
 2   return new List<Client> {
 3     new Client {
 4       ClientId = "mvc",
 5       ClientName = "MVC client",
 6       AllowedGrantTypes = GrantTypes.HybridAndClientCredentials,
 7       ClientSecrets = { new Secret("mysecret".Sha256()) },
 8       AllowedScopes = {
 9         "openid", "profile", "email", "phone", "ar.profile", // identity res
10         "users" // API resource
11       },
12       ...
13     },
14     new Client { ... }
15   }}
```

Program 5.3: Definition of clients and their allowed scopes which refer to identity and API resources; line 9 defines identity resources and line 10 defines the API resource to access the users endpoint

Identity and access tokens are transferred in form of JSON Web Tokens (JWT) [JBS15] between the main service and the identity service. JWTs contain the issued claims and are self-contained. They are signed by the identity service which allows other services to validated the integrity of tokens using the identity service's public key. When a client accesses a protected microservice, it attaches the access token to the HTTP Authorization request header using the Bearer scheme. Since the microservice can validate the token without sending a request to the identity service, this facilitates a decoupling from the identity service. Program 5.4 shows an access token payload for a request sent from the main service to the users endpoint.

As a result of the decoupling from the identity service, microservices can process requests even when the identity service is not available. Thus, an application user does not notice any disturbance if he is already logged in and as long their access token does not expire.

```
{
  "nbf": 1524908056,
  "exp": 1524911656,
  "iss": "http://localhost:5105",
  "aud": [ "http:// localhost:5105/resources", "users", ... ],
  "client_id": "mvc",
  "sub": "fd642ace-48a1-4903-9879-c583f35931d3",
  "auth_time": 1524908054,
  "idp": "local",
  "ar:usertype": "SiteAdmin",
  "scope": [ "openid","profile","email","phone","ar.profile","users", ... ],
  "amr": [ "pwd" ]
}
```

Program 5.4: The payload of an access token in form of a JWT: `sub` is the id of the logged-in user and as `ar:usertype` indicates, they are allowed to to execute actions which require SiteAdmin rights. The URI http://localhost:5105 represents the identity service as the issuer (`iss`) in a local development environment.

5.3.2 A common API service for all clients

In the next step, the view rendering component is extracted from the main service. Therefore, the view logic and view models are moved into a new ASP.NET MVC service. The main service needs to expose an API so that the MVC service is able to call the business logic and request data required for view rendering. Since the main service did already provide a REST API for the mobile app, this is also considered designing the new REST API. Therefore, the existing REST API and the existing business logic, which was previously provided to the view rendering component, are merged. The merging process has shown that some functionality was implemented twice; this has been cleaned up in order to provide a consistent REST API. Figure 5.4 depicts the application architecture after extracting the MVC service.

To allow the REST API to constantly evolve, versioning is introduced. Thus, the REST API is independent of the web application, the mobile app or any other consumer. Every route of the main service contains the major version number; when breaking changes are necessary, the version has to be incremented. To make sure the already released mobile app does not break with the introduced versioning, the old routes need to remain available until the mobile app's current version is not in use anymore. ASP.NET Core allows annotating controller methods with multiple routes, which eases the transition to the new versioning approach. Program 5.5 shows how multiple routes can be specified.

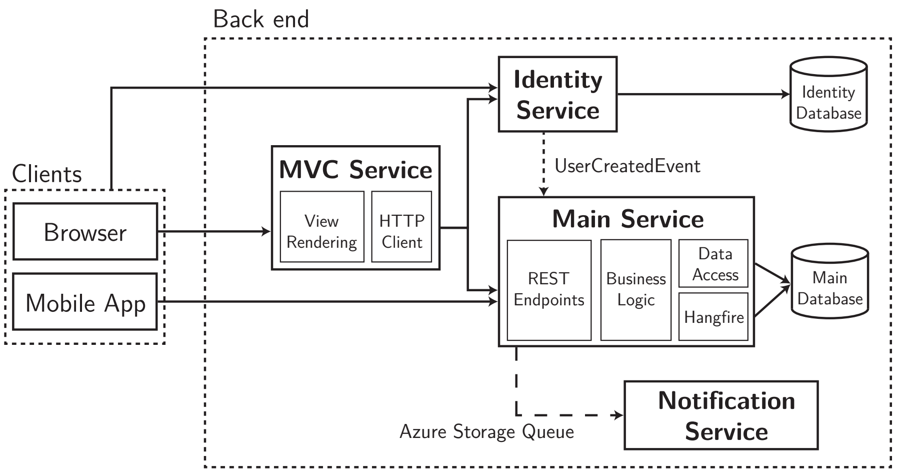


Figure 5.4: Separation of main service and MVC web application

```
[Route("api/v1/Tasks")]
public class TaskController : Controller {
  [HttpPut("status")]                  // resolves to: api/v1/Tasks/status
  [HttpPost("~/api/task/changestatus")]  // legacy route
  public async Task<IActionResult> ChangeStatus(TaskChangeModel model) {
    ...
  }}
```

Program 5.5: Multiple route definitions on a single controller method; controller routes are extended by method routes, unless prefixed with tilde (˜)

Fault tolerance

To mitigate failures that will eventually occur in a microservices architecture, several patterns are applied. The MVC service uses HTTP calls to communicate with the identity service and the main service. These HTTP calls are a potential source of error and consequently need to be fault-tolerant. Polly[7] is a library that allows to wrap HTTP calls with fault-handling mechanisms to increase application resilience. As Program 5.6 shows, Polly allows to specify different fault-handling approaches which can also be combined to a policy wrap. The following policies are used to handle failures:

- Line 2 defines a timeout policy. Each request that takes longer than three seconds will fail.
- Lines 4 to 8 define a retry policy that handles all failures but excep-

[7]http://www.thepollyproject.org

```
 1 public PolicyWrap CreatePolicies() {
 2   var timeoutPolicy = Policy.TimeoutAsync(TimeSpan.FromSeconds(3));
 3
 4   var retryPolicy = Policy
 5     // don't retry on inner circuit breaker exception
 6     .Handle<Exception>(e => !(e is BrokenCircuitException))
 7     .RetryAsync(1, (ex, attempt) => { // retry once
 8       _logger.LogError($"Retrying ({attempt}). {ex.Message}"); });
 9
10   var circuitBreakerPolicy = Policy
11     .Handle<Exception>()
12     .CircuitBreakerAsync(
13       exceptionsAllowedBeforeBreaking: 5,
14       durationOfBreak: TimeSpan.FromSeconds(10),
15       onBreak: (ex, delay) => { _logger.LogError("Breaking ..."); },
16       onHalfOpen: () => { _logger.LogError("Setting half-open"); },
17       onReset: () => { _logger.LogError("Resetting: successful call"); });
18
19   var bulkheadPolicy = Policy.BulkheadAsync(TotalBulkheadCapacity / TotalEndpoints);
20
21   return PolicyWrap.WrapAsync(timeoutPolicy, retryPolicy, circuitBreakerPolicy,
       bulkheadPolicy);
22 }
```

Program 5.6: Policies used by MVC service for HTTP requests to other services

tions due to a broken circuit. There is only one retry applied, because otherwise an application user may need to wait too long for a response from the MVC service. In case a retry is executed, the attempt is logged with the previously occurred exception.

- Lines 10 to 17 define a circuit breaker policy. When five consecutive calls have thrown exceptions, the circuit breaker will break for ten seconds. In these ten seconds every call will fail without going through to the receiving service. After this period, the result of the next call determines if the circuit breaker transitions into open or closed state again.
- Line 19 defines a bulkhead policy. `TotalBulkheadCapacity` is the number of maximum open connections in parallel and is divided by the total endpoints a service is making calls to. Therefore an unresponsive endpoint does not lead to a saturation of available connections.
- Line 21 shows how these policies are combined into a policy wrap. The policy wrap is then used by the MVC service to execute HTTP calls. When wrapping policies, the order is important: The leftmost policy, in reading order, wraps the following one, which then again wraps the following one, etc.

The Polly library is also used within the message bus abstraction to handle event subscribing and publishing in a fault-tolerant way. When an exception occurs that is related to an unreachable message bus, the action is retried with exponential backoff.

Coupling between identity service and main service

As a first step to extract the identity service from the main service, the main service synchronously queries the identity service when it needs user information, like first and last name or the email address. Synchronous communication is easier to implement because of its similarity to inter-process calls within a monolith.

But, as explained in Subsection 4.4.1, microservices should only communicate asynchronously in order to achieve a better decoupling – an exception are services that are only responsible for aggregating data from multiple other services to render views, like the MVC service. While synchronous communication decouples services to a certain degree, because it allows independent development and deployment, only asynchronous communication can increase an application's overall resilience and mitigate the impact of unavailable services. To remove the synchronous calls, the required data could be replicated to the main service. On each change to its data, the identity service would need to propagate these changes to the main service which then also updates its data replication.

5.3.3 Migrating the notifications worker service

The last step of the prototypical transformation into a microservices architecture is the refactoring of the notifications service. The notifications service is responsible for sending emails and text messages to volunteers, event managers, etc. For this purpose it uses third-party software like Twilio and SendGrid, and sends HTTP requests to these services.

In the initial, monolithic architecture, the notifications service is already implemented as a separate service. It is developed based on the Azure WebJobs SDK targeting the .NET Framework and is bound to a queue that provides the messages to be processed. This queue is an Azure Queue Storage, a product provided by Microsoft within the Azure Cloud.

Since worker services all operate on a shared backlog of work, they are very well suited to scale for improved throughput of work and also for improved resilience [New15, pp. 220 sq.]. In order to remove the dependency on the Azure Cloud, the notifications service is refactored into a .NET Core service that subscribes to the event bus instead of the Azure Queue Storage. As an alternative, the worker service could also be developed with Azure Functions[8], which is a serverless compute service; but this would have the disadvantage of a vendor lock-in to the Microsoft Azure cloud.

5.4 Discussion

This section is evaluating the transformation of the application into a microservices architecture following an incremental process. Concerning the development of allReady as an open source application, possible benefits and drawbacks of a microservices architecture are identified.

The refactoring of allReady from a monolithic architecture into multiple independent microservices has shown that this is a non-trivial task and requires a very good understanding of the business domain. Although the application is only moderately large of size, it was possible to split it into several individual services in order to enable horizontal scaling. The extraction of the MVC service resulted in a relatively high effort, since every method call has to be exposed via a public REST API. The MVC service consumes this API via a HTTP client library which also requires additional error handling. As mentioned, the complexity of the business functionality is still comprehensible. Therefore, the main service has not been split any further. This avoids the problem of premature decomposition which might become a hindrance to further development when requirement changes involve multiple services [New15, pp. 33 sq.].

The consolidation of the authentication and authorization processes reduces maintenance effort because no custom tokens – which were used for the mobile app – need to be managed in the database anymore. Moreover, it eliminates the necessity for duplicated REST API routes; the mobile app as well as the web application had separate routes since the token validation was only executed for the mobile-app routes.

The utilization of self-contained JWTs in order to attach the user identity to the request brings another improvement: because the token contains all necessary information about the user's identity, no database query is required to retrieve this information, as opposed to the custom token validation, where

[8]https://azure.microsoft.com/en-us/services/functions

a user lookup was required.

All microservices share a common set of functionality they have to fulfill in order to accomplish application resilience. They need to enable fault-tolerant communication (synchronous as well as asynchronous), persist logging information, handle access tokens, etc. Microservices have been around for some time now and from a technological perspective, many supporting libraries and frameworks have evolved for different languages and environment. Even though .NET Core is relatively new, useful open source libraries are available. In particular, Polly reduces the effort to implement fault-handling. Additionally, IdentityServer4, which is also open source, enables fast implementation of OAuth 2.0 and OpenID Connect workflows. Several extension libraries for IdentityServer4 allow quick integration of new services into the authentication and authorization system.

For open source software, and especially for allReady, it might be difficult to find voluntary contributors. A complex setup of the entire application solution and also the large application domain can deter new contributors. To contribute to allReady, one has to setup the entire application with all of its dependencies. Setting up a local build environment for the mobile app is required to manually test the app against the back end. Here, consumer-driven contracts, as mentioned in Subsection 4.4.3, would help back end developers to ensure the consumed API is working as intended.

Moreover, by dividing the monolithic application into independent microservices, contributors are not required to have knowledge of the whole architecture. This lowers the entry barrier and as a result could increase the number of contributions. For example, if one's strengths are HTML, CSS and JavaScript, they can focus on implementing the MVC service without having to comprehend the complexity of the main service.

When creating new microservices by taking functionality out of the monolith, in theory, the technology can be chosen without restrictions. But if the team decides to switch to a different technology stack than the one used by the monolith, code cannot simply be moved but has to be rewritten. This requires more effort, especially if a library or framework was utilized in the monolith, for which no technological equivalent exists.

Regarding the attraction of new contributors to the open source application, a heterogeneous technology stack brings a wider possible audience. Using various languages for different services results in a greater coverage of possible developers' core skills. In contrast, since the number of languages a contributor is skilled in is limited, a homogeneous technology stack allows contributors to switch to other services if it is required. With a heterogeneous stack – depending on the degree of heterogeneity – this is only possible to a very limited extent.

Part II

Operations

Chapter 6

Running Containerized Microservices in the Cloud

In order to leverage the full potential of microservices, running the application on cloud infrastructure is essential. Since microservices excel at a large scale, fast provisioning of infrastructure is indispensable. Besides the flexible provisioning of cloud resources within a very short time, cloud computing comes with several other advantages. Due to the automatic provisioning of resources performed by the cloud vendor, companies don't have to manage and maintain infrastructure. They can achieve cost savings by only utilizing the resources needed to handle the current workload. Furthermore, rapid provisioning of new infrastructure allows to handle exceptional peaks in the application's workload. The resulting benefits of cloud computing are not only limited to hardware: on a software level, for instance, it is important to maintain software dependencies or apply critical security patches. By delegating this responsibility to the cloud vendor, companies do not need in-house expertise.

A key concept to achieve fast provisioning of cloud resources is virtualization. In contrast to common hypervisor-based virtualization, operating system-level virtualization schedules containers that are a more lightweight abstraction and share the host's kernel. This facilitates a smaller resource footprint but comes with deficiency in isolation. The industry-leading container technology is Docker, as 79 percent of all deployments use Docker as their runtime [Sys19]. A service deployed within a Docker container is self-contained and bundles all its required dependencies. This isolation allows for loose coupling between containers and therefore, aligns well with a microservices architecture. Because a container comes with all its dependencies, a microservice can be implemented using any framework or programming language. The image-based approach of Docker enables the definition of infrastructure as code within a Dockerfile.

But Docker doesn't solve the problem of running a container-based applications across multiple hosts. This chapter discusses how workloads can be distributed in a cluster consisting of several servers or virtual machines. Systems that solve this problem are called container orchestration systems. Due to the focus of this thesis on Microsoft technology, container orchestration solutions available in the Azure cloud are evaluated and compared. The basis for this purpose is the allReady microservices application from Chapter 5 which is operated on each solution suitable for container orchestration.

6.1 Introduction to container orchestration

Container orchestration is the process of managing containers running on a cluster that spans over multiple nodes. These nodes could be physical servers or also virtual machines and host a variate of different containers. A container orchestration platform abstracts away these nodes and enables the interaction of containers within an application independent of their location.

Khan [Kha17] defines container orchestration platforms as "a system that provides an enterprise-level framework for integrating and managing containers at scale." He names the following seven key capabilities a container orchestration platform has to implement:

- **Cluster state management and scheduling:** This part of the orchestration observes the cluster state and allows other system components to react to changes by notifying them. It detects when a node in the cluster becomes unavailable and counteracts by rescheduling the affected containers on healthy nodes in order to ensure the correct number of containers are running. The goal of scheduling is to balance workload between the nodes and optimize resource usage. To achieve this, container orchestration creates, deletes or moves containers to a different node, taking into account the resource requirements of a container (e.g. CPU time or memory). Also, the orchestration has to satisfy defined constraints or affinities between containers. In order to avoid starvation of co-hosted containers, it limits the resources available to a container.
- **Providing high availability and fault tolerance:** A high fault tolerance can be achieved by adding redundancy. Multiple redundant replicas

of a service or orchestration component mitigates the problem of a single point of failure. Implementing the load balancing pattern allows to share the workload between service replicas. This improves performance and reduces the risk of a service instance failing under heavy load. The orchestration detects when a service or container fails. In this case the goal is to keep the application in a functioning, albeit degraded, state.

- **Ensuring security:** To ensure a high level of security it is inevitable to keep a container's attack surface as small as possible. As already mentioned, containers on the same node share the host's kernel. Therefore, containers with root access are a security risk due to the weaker isolation. In addition, a container orchestration system needs to provide a component for secret management to store container or service specific data and restrict access to those.
- **Simplifying networking:** Docker only enables containers to communicate if they are located on the same host. Since a container orchestration system spans across multiple nodes it has to provide functionality for containers to communicate regardless of which nodes they are located on. This requires the nodes to have ports allocated for all containers they are hosting. At scale this needs to be done automatically.
- **Enabling service discovery:** Containers are created, deleted and moved with a high frequency which results in dynamically changing network locations. To make communication between containers possible, orchestration has to provide functionality for service discovery. This can be solved with the service registry that stores the network location of all containers. There are two possible approaches to find a containers location: client-based discovery and server-based discovery. With client-based discovery the client itself queries the registry and is then responsible to perform the load balancing. This method introduces a coupling between client and service registry. With server-based discovery, the client makes request via a dedicated load balancer which subsequently queries the service registry. This removes the need for client-specific discovery logic and thus, facilitates a loose coupling between client and registry. To avoid the need for maintenance, the load balancer should be part of the orchestration system.
- **Making continuous deployment possible:** In a microservices architecture, the deployment of a service is a frequent action. Thus, a orchestrator has to embrace the process of continuous deployment and ease the handling of a deployment pipeline. Such a pipeline consists of multiple stages: commit, build, staging, production and feedback loop. A container orchestrator should support different environments for different stages. Also, it should provide roll-out strategies for upgrading services without downtime.

- **Providing monitoring and governance:** An orchestration system should facilitate extensive monitoring of running containers and the cluster infrastructure. In particular, this comprises tracing requests, monitoring resource consumption and logging.

6.2 Requirements and design decisions

The goal is to explore the capabilities of the Azure cloud to run containerized microservices. Azure offers the following products to run containers [Micc]:

- **Container Instances:** Use cases are infrequent or on-demand workloads. Container Instances are not designed to run a complete microservices application. Furthermore, they are not available in all Azure regions and come with limited available resources, which are 4 cores and 16 GB of RAM for a single container [Mic19c].
- **App Service:** Also App Service is not a good fit for containerized microservices because independent services cannot be managed and deployed separately. Additionally, at this time it does not support applications that consist of multiple containers for production workloads [Mic19l].
- **Batch:** Batch is a product to run jobs at a large scale but does not allow the coordination of microservices.
- **Azure Kubernetes Service:** AKS is a fully managed Kubernetes cluster. Kubernetes is an open source system for container orchestration. AKS takes care of the underlying infrastructure Kubernetes is running on.
- **Service Fabric:** Service Fabric is a system developed by Microsoft with the intention of running distributed applications. It allows different programming models: applications can either be integrated into the Service Fabric platform by utilizing the provided API and framework, or they can be deployed within a Docker container that is managed by Service Fabric.

Only Azure Kubernetes Service and Service Fabric meet the requirements for orchestrating a microservices application and thus are included in the comparison. Both platforms are managed by Azure. This reduces the complexity of maintaining cluster infrastructure and allows to reduce operational cost. Both Azure Kubernetes Service and Service Fabric still provide access to the underlying virtual machines. This can be helpful for debugging scenarios.

Only Service Fabric supports Linux as well as Windows containers. Currently, Azure Kubernetes Service only supports orchestrating Linux containers. Microsoft is working on supporting Windows containers but no date for general availability is communicated [Bro19]. For the evaluation and comparison, allReady is adapted to Service Fabric on Linux, Service Fabric on Windows and Azure Kubernetes Service on Linux.

6.2.1 Requirements on the orchestration level

Each platform provides its own ecosystem and offers a number of solutions to implement certain tasks. As a limitation, only mature components should be used for the platform adaption. An exception to this rule is if no alternatives are available. Only then components in preview state are acceptable. In order to keep the coupling between application and orchestration to a minimum, orchestration-specific changes to the application are avoided as far as possible.

Application-level infrastructure components – like message bus, databases or reverse proxies – should be operated as services in the cluster. A self-contained architecture simplifies the application's setup on new environments; this is particularly beneficial for local development environments. For production environments, these components could be replaced with solutions that are hosted by the cloud provider that offers a service-level agreement. Exemplary infrastructure components utilized by allReady are a RabbitMQ message bus or Redis databases to share temporary state between service instances. An exception to these in-cluster operated components are SQL databases which are used as the primary data storage. In this case an Azure hosted SQL server is utilized.

Another requirement on the orchestration level is to take the load of HTTPS termination off the individual microservices. An API gateway should be responsible for terminating the incoming HTTPS connections and then forward the requests to the respective services.

To keep the comparison's complexity in a comprehensible state, each container orchestration system only uses one type of virtual machine for its cluster.

```
var redis = ConnectionMultiplexer.Connect(redisConnectionString);
var protectionBuilder =
  services.AddDataProtection(options => {
    options.ApplicationDiscriminator = "allready-mvc";
  });
protectionBuilder.PersistKeysToRedis(redis, "DataProtection-Keys");
```

Program 6.1: Simplified code to configure a shared Redis database to store CSRF tokens.

```
 1 public void Configure(IApplicationBuilder app, ...) {
 2   app.UseForwardedHeaders();
 3 }
 4
 5 public void ConfigureServices(IServiceCollection services) {
 6   services.Configure<ForwardedHeadersOptions>(options => {
 7     options.ForwardedHeaders =
 8       ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;
 9     options.KnownNetworks.Clear();
10     options.KnownProxies.Clear();
11   });
12 }
```

Program 6.2: Middleware configuration in `Startup.cs` to apply forwarded headers; lines 9 and 10 allow requests from all addresses. In a production scenario a whitelisting approach is recommended.

6.2.2 Requirements on the application level

Since the notifications service has been reworked using .NET Core, all microservices are based on .NET Core. In combination with Docker containers, the modular structure of .NET Core allows to significantly reduce the container image size by only including the functionality that is actually used.

Scaling a service within a microservice architecture often requires its instances to share state. A trivial example would be service instances sharing a user's session so any instance can serve requests by this user. The following application level changes are made to support multiple parallel instances:

- Both MVC service and identity service use Cross-Site Request Forgery (CSRF) tokens to validate forms submitted by the user. ASP.NET Core provides functionality to manage these tokens in a shared storage. Both microservices use their own Redis database as a shared storage. Program 6.1 shows the configuration of Redis as a token store.

- The identity service requires operational data (e.g. issued tokens) to be shared between its instances. Identity Server 4 itself does not provide functionality to persist this data to Redis but provides the interface `IPersistedGrantStore` that can be implemented for any storage mechanism [AB]. The package `IdentityServer4.Contrib.RedisStore`[1] already implements this for Redis.
- Since a reverse proxy handles the effort of HTTPS termination and then forwards requests via HTTP, the information of the original protocol must be transmitted to a microservice in a different way. Furthermore, information about the client that made the request to the application might be necessary and needs to be passed on. Reverse proxies commonly forward this information via HTTP headers. The receiving services pick up these headers and apply them to the request. ASP.NET Core provides a middleware to enable this; Program 6.2 shows how to configure the middleware. Per default, forwarded headers are only accepted from proxies on loopback addresses. In a distributed application, this restriction has to be removed.
- Services behind a reverse proxy might be publicly accessible via a sub path relative to the domain. This sub path can be used by the reverse proxy to determine which microservice it should route the request to. An example would be `www.allready.org/identity/login` to access the identity service' login route. When the microservice receives requests from the reverse proxy, it is not aware of its sub path. Therefore, its routing component will fail trying to find the intended `/login` route because the request points to `/identity/login`. To solve this problem, ASP.NET Core provides the `UsePathBase` middleware that strips the sub path before routing to a controller and also reapplies it when rendering relative hyperlinks in views. Which sub path the middleware should process is defined via an environment variable.

[1]https://github.com/AliBazzi/IdentityServer4.Contrib.RedisStore

Chapter 7

Comparison of Azure Kubernetes Service and Service Fabric

This chapter discusses the differences and similarities of Azure Kubernetes Service and Service Fabric. At first, the criteria for this comparison are explained. Then, each criterion is discussed one after another emphasizing the practical challenges, advantages and, disadvantages each orchestration platform brings. This structure intends to allow the reader to better identify the differences between the rather comprehensive and complex platforms. Prototypical implementations for each orchestration platform are shown using the example of the allReady application. This comparison is based on AKS version 1.14.8, Service Fabric on Linux version 6.5.466.1 and Service Fabric on Windows version 6.5.664.9590.

7.1 Criteria

The characteristics defined by Khan [Kha17] serve as a basis for the comparison and evaluation of the container orchestration systems. However, the monitoring aspect will not be addressed in this thesis because both platforms can be integrated with numerous monitoring solutions. Including these would considerably exceed the scope.

As an additional criterion, load tests are run against each cluster. For this purpose, a workload is generated that approximates a realistic usage. To achieve this, different user personas with their individual workflows are specified. The resulting performance metrics are then compared.

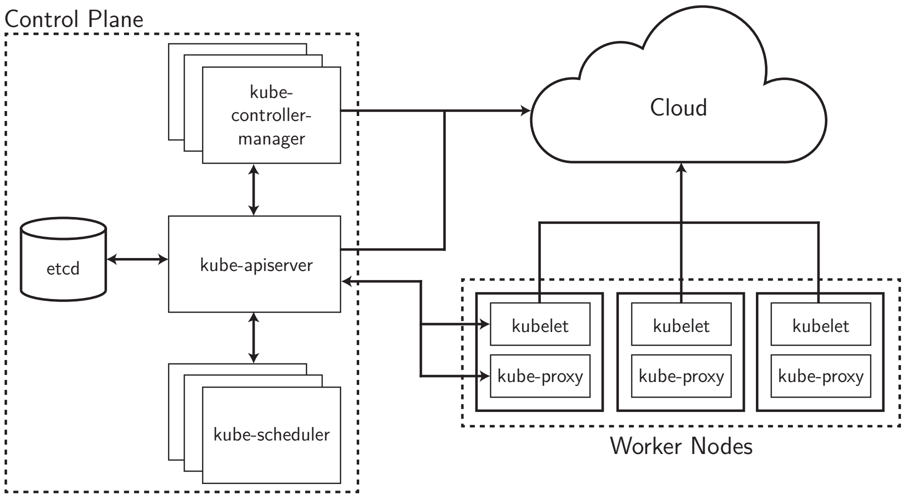


Figure 7.1: Kubernetes cluster architecture; adapted from [Kub19d]

7.2 Azure Kubernetes Service

Azure Kubernetes Service (AKS) is a product by Microsoft that simplifies deploying the deployment and operation of a Kubernetes cluster on Azure infrastructure. Kubernetes itself is an open-source orchestration system for containers and is managed by the Cloud Native Computing Foundation (CNCF). Kubernetes version 1.0 was release on 21 July 2015 [Vau15] and has since become the most used container orchestration system [McA+; Sys19].

7.2.1 Cluster architecture

A Kubernetes cluster consists of two different node types: master nodes that manage the cluster and worker nodes that run the containers. Figure 7.1 gives an overview of the Kubernetes architecture. A cluster's master nodes are called the control plane; this control plane consists of the following components [Kub19j]:

- **kube-apiserver:** This central component provides the API to interact with the cluster. It is used by all other Kubernetes components for operations on the cluster state.
- **etcd:** etcd is a highly-available key-value store that allows to be distributed across nodes. Kubernetes uses etcd to persist the entire cluster

state and configuration. On top of that, different Kubernetes components use etcd's watch functionality to listen for state changes [Etc19].

- **kube-scheduler:** The smallest logical resource in Kubernetes is called a pod and consists of at least one container. When a new pod is created, kube-scheduler decides on which node it will be scheduled. Several factors are taken into account in this decision-making process. Some examples are the resource utilization of nodes, constraints in required resources or affinity between pods.
- **kube-controller-manager:** This component runs controllers which manage a certain aspect of the cluster's state. When a state change is desired, it uses the kube-apiserver to accomplish that. For instance, the Replication Controller ensures that the correct number of pod replicas are running. Another example is the Node Controller that holds a virtual representation of each node and keeps it in sync with the cloud provider. That way, it can detect if a node has been deleted when it becomes unresponsive.

Worker nodes within the Kubernetes cluster are responsible for running the pods encapsulating containers. The workload coordination is done by the control plane. Therefore, it instructs the worker nodes which pods they need to run. A worker node has the following components:

- **kubelet:** This is the agent that manages the running pods. Basically, it communicates with the control plane via the kube-apiserver which provides the kubelet with a set of pod specifications. The kubelet is responsible that all containers described in the pod specifications are in a healthy state.
- **kube-proxy:** This network proxy is used to enable communication between pods within the cluster. An arbitrary number of pods can form a service that is reachable via a cluster IP address. To accomplish this, kube-proxy manages network rules using iptables [Kub19q].
- **Container runtime:** Besides the industry leading Docker runtime, Kubernetes also supports containerd[1], rkt[2] and CRI-O[3]. Additionally, Kubernetes specifies the Container Runtime Interface (CRI). The CRI enables the usage of any container runtime that implements the interface.

Complimenting the above components, addons provide further cluster functionality. Some exemplary addons are: a DNS server to simplify communi-

[1]https://containerd.io
[2]https://coreos.com/rkt
[3]https://cri-o.io

cation with services, a dashboard UI to manage Kubernetes resources, tools for container resource monitoring, or a central log store for container logs.

AKS is a platform that provides a managed Kubernetes cluster and is available for production usage since June 2018 [Bur19]. It takes on the management of the control plane and provides high availability by replicating the master nodes. It is not possible to directly access the control plane. However, AKS provides an interface that allows upgrading Kubernetes to a newer version or updating the number of worker nodes. A worker node is a virtual machine in the Azure cloud. Currently, AKS supports Kubernetes version 1.14.8 and thus is two minor versions behind the most recent Kubernetes 1.16 release. Even though Kubernetes supports different container runtimes, in an AKS cluster the only available runtime is Moby. Moby is the open source upstream project of Docker and therefore runs Docker containers [Mic19f].

7.2.2 Platform model

The Kubernetes API follows a declarative approach. It uses Kubernetes objects for all kinds of API resources. These objects represent the complete state of the cluster and are persisted in the etcd store. Kubernetes objects are declared in a YAML manifest file. In order to update the cluster state, a Kubernetes object is manipulated and then sent to the Kubernetes API with the intent to update the state.

To give an overview how Kubernetes objects work, the fundamental objects to create a replicated service are explained:

- **Pod:** As already mentioned above, the smallest building block in Kubernetes is a pod which serves as a unit of deployment. In most use cases, it consists of a single container and represents a concrete microservice. To run multiple instances of a microservice, the pod can be replicated using a deployment object explained below. Additionally, it is also supported to bundle multiple containers within one pod if they require tight coupling. In this case, all containers of a pod are scheduled on the same node and share resources [Kub19n].
- **Deployment:** A pod itself should not be managed manually because if it fails, it won't be rescheduled. Generally, a pod should be considered ephemeral. In order to ensure that a certain number of pod replicas are running in the cluster, a deployment object can be used. The declaration of a deployment contains the pod specification and the number of replicas of this pod. A deployment is also used to roll out updates to the pods' state, e.g. when the container image has been updated;

```
 1 apiVersion: apps/v1
 2 kind: Deployment
 3 metadata:
 4   name: allready-identity
 5 spec:
 6   replicas: 3
 7   selector:
 8     matchLabels:
 9       app: allready-identity
10   template:
11     metadata:
12       labels:
13         app: allready-identity
14     spec:
15       containers:
16       - name: allready-identity
17         image: allreadyacr.azurecr.io/allready.identity@sha256:e375d7...
18         ports:
19         - containerPort: 80
20         env:
21         - name: BasePath
22           value: /identity
```

Program 7.1: A deployment object in YAML format describing the allReady identity service; line 6 defines the number of pod replicas, lines 14 to 22 show the definition of the pod object that consists of a single container, line 13 (highlighted in blue) defines a label which is later matched by the service object in Program 7.2

or to create more replicas in case of an increasing load [Kub19f]. Program 7.1 shows the declaration of a simplified deployment object.

- **Service:** Each pod has its own IP address assigned but since pods are created and removed frequently, a mechanism is required that allows pods to be easily discoverable. In Kubernetes a service object abstracts a set of pods and keeps track of their network locations. It acts as an intermediary to provide a discovery mechanism to other microservices without them requiring knowledge about the specific pods. In other words, microservices send requests to a service object which then load balances the request between the abstracted pods. To define the cohesiveness between service and pods, so called label selectors are used [Kub19q]. This allows a service to span multiple deployments. Program 7.2 shows the declaration of a service object.

```
 1 apiVersion: v1
 2 kind: Service
 3 metadata:
 4   name: allready-identity
 5 spec:
 6   type: ClusterIP
 7   ports:
 8   - port: 80
 9   selector:
10     app: allready-identity
```

Program 7.2: A service object in YAML format that enables the allReady identity service to be discovered within the cluster; line 9 and 10 show the label selector that matches the pods created by the deployment from Program 7.1

7.3 Azure Service Fabric

Service Fabric is a distributed platform that facilitates the development of applications following a microservices architecture. It is developed by Microsoft and used internally to operate infrastructure services in the Azure cloud. Furthermore, Service Fabric is the underlying foundation for several Azure products like SQL Server, Cosmos DB or Cortana.

In 2018, Microsoft made the platform available as open source; the development is still managed by Microsoft's Service Fabric team [Ser18]. Although community contributions are allowed, only a total of 64 commits have been made to the public GitHub repository since it became open source [Git].

Service Fabric supports several different programming models to develop distributed applications. The first two of these approaches are deeply integrated into the Service Fabric platform and its API [Mic17f]:

- **Reliable Services:** The Service Fabric platform provides the Reliable Services framework to implement stateless and also stateful services. The advantage of stateful services is that they don't require an external storage to persist their state because state is persisted within the service itself. To accomplish this, Service Fabric offers Reliable Collections which replicate state across the cluster and allow high availability.
- **Reliable Actors:** Based on Reliable Services, the Reliable Actors framework implements the actor design pattern.

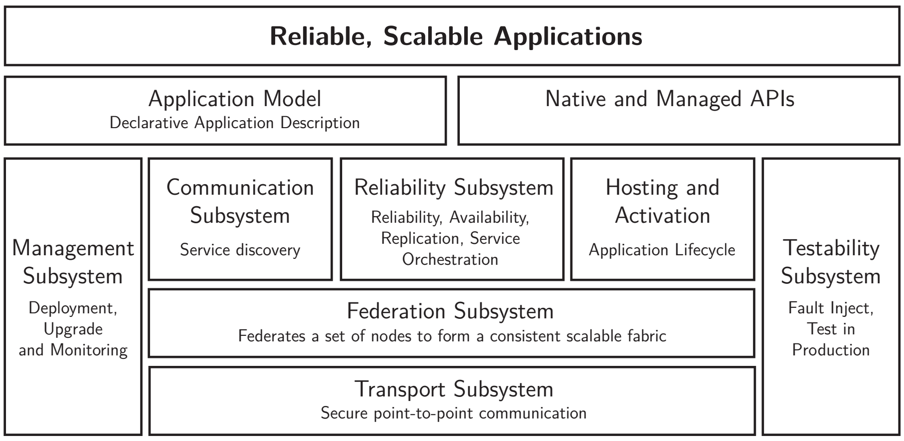

Reliable, Scalable Applications
Application Model
Declarative Application Description
Native and Managed APIs
Management Subsystem
Deployment, Upgrade and Monitoring
Communication Subsystem
Service discovery
Reliability Subsystem
Reliability, Availability, Replication, Service Orchestration
Hosting and Activation
Application Lifecycle
Testability Subsystem
Fault Inject, Test in Production
Federation Subsystem
Federates a set of nodes to form a consistent scalable fabric
Transport Subsystem
Secure point-to-point communication

Figure 7.2: Service Fabric cluster architecture [Mic17e]

- **Containers**: Service Fabric is also able to orchestrate containers. This does not require the service to integrate any Service Fabric related logic and therefore, decouples the operated microservices from the orchestration platform.
- **Guest executables**: In addition, Service Fabric can run any executable. These executables don't use the Service Fabric API.

This thesis focuses on the orchestration of platform-independent, container-based microservices; therefore, Reliable Service, Reliable Actors and guest executables won't be discussed in detail.

7.3.1 Cluster architecture

Service Fabric can be deployed to the Azure cloud and also to on-premises infrastructure. Its architecture differs from Kubernetes, as the nodes are not distinguished between master nodes and worker nodes. All nodes are equal in regards to the running Service Fabric components. As illustrated in Figure 7.2, a node is described by the following components – often called subsystems – that build on each other [Kak+18]:

- **Transport subsystem:** The underlying component for all other subsystems is the transport subsystem. This Service Fabric internal channel secures the communication of the nodes within the cluster and also the communication between cluster and clients. To prevent unauthorized access, the transport subsystem either uses X509 certificates or

Windows Security.

- **Federation subsystem:** Service Fabric has the goal to enable strong consistency in its foundation and thus remove this complexity from the application layer on top of it. Built atop of the transport subsystem, the federation subsystem implements the detection of node failure, leader election and routing. A peculiarity of Service Fabric's federation subsystem is that it decouples the detection of a failing node from the final decision about the failure. This solves the problem of contradictory decisions in a distributed system. In order to detect failure, the federation subsystem utilizes a virtual ring, called SF-Ring. All nodes are mapped onto this ring. Each node monitors a number of its predecessors and successors in regards to their order in the ring. These tracking relationships are bidirectional, meaning in a pair of nodes both monitor one another. In this monitoring process, a node sends heartbeat messages – so called lease renewal requests – to all its monitors, and then receives back the acknowledgement. Whether a node detected as failed is excluded from the ring, is decided by the arbitrator group. This group is formed by a number of independent nodes. In order to confirm a node failure, a quorum in the arbitrator group must acknowledge it.
- **Reliability subsystem:** The three main components of the reliability subsystem are: the failover manager, the placement and load balancer, and the naming service. The failover manager is responsible for creating and upgrading services. It gets informed about each node's resource utilization and communicates with the placement and load balancer (PLB). The creation of a new service replica is initiated by the failover manager, and then delegated to the PLB which determines the node the replica is created on. The PLB takes into account several factors, such as constraints or affinity. In case of a node failure, the underlying arbitrator notifies the failover manager. Subsequently, the failover manager initiates the replica recreation process. The naming service is responsible to resolve a service name to its network location.
- **Management subsystem:** This subsystem provides an interface to interact with the cluster and change its state. Within a Service Fabric cluster, microservices are deployed, updated or removed using the cluster manager component. To manage an application's lifecycle across the cluster, it interacts with several subsystems: it instructs the failover manager to create new services, or delegates the service monitoring to the health subsystem. Another component of the management subsystem is the health manager that aggregates all health information into a centralized store.
- **Hosting subsystem:** On each node in the cluster, the hosting subsystem is executed. Upon instructions from the cluster manager, it manages

the lifecycle of a service on the specific node it is operating on.

- **Communication subsystem:** The communication subsystem provides the naming service that allows services to resolve other services' location.
- **Testability subsystem:** This component provides tools for testing applications. By simulating faults of different kinds, the behavior of degraded applications can be evaluated.

To visualize a cluster, the Service Fabric Explorer is provided. This is a web application that shows health information about infrastructure and microservices. It also supports a small set of administrative operations.

7.3.2 Platform model

On top of these subsystems, Service Fabric provides a declarative approach to deploy microservices. This is called the application model. The model is specified using manifest files in XML format. The logical architecture of the application model differentiates between applications and services. Program 7.3 and Program 7.4 are simplified example manifests for an application type and a service type respectively.

An application is composed of several services and also has to specify which service instances to create when the application is initialized – these are so called default services. If a service type is not specified as a default service, it has to be created manually. However, since most services should always be available and a manual creation is not preferable with regard to scalability, the explicit declaration of default services is mandatory. This results in verbose manifest files that are harder to maintain.

Both application type as well as service type allow multiple co-existing instances in a cluster. Service instances can be scaled easily by updating the replica count; in order to increase the instances of an application type, new instances with different names have to be deployed one by one.

A service runs independently of other services and consists of three parts: a code package, a configuration package and a data package. The code package specifies the service entrypoint that is executed at startup; for containerized services, this points to the Docker image. Additionally, a setup entrypoint can be specified which is commonly used to execute scripts with higher privileges. The configuration and data packages can be ignored because a container is self-contained and must not depend on external data. To inject

```xml
 1 <ApplicationManifest ApplicationTypeName="Identity.SFType" ...>
 2   <Parameters>
 3   <Parameter Name="BasePath" DefaultValue="" />
 4   </Parameters>
 5   <ServiceManifestImport>
 6     <ServiceManifestRef ServiceManifestName="IdentityPkg" ... />
 7     <EnvironmentOverrides CodePackageRef="Code">
 8       <EnvironmentVariable Name="BasePath" Value="[BasePath]" />
 9     </EnvironmentOverrides>
10     <Policies>
11       <ContainerHostPolicies CodePackageRef="Code">
12         <RepositoryCredentials AccountName="allreadyacr" Password="*" ... />
13         <NetworkConfig NetworkType="Open" />
14       </ContainerHostPolicies>
15     </Policies>
16   </ServiceManifestImport>
17   <DefaultServices>
18     <Service Name="Identity" ServiceDnsName="allready-identity" ...>
19       <StatelessService ServiceTypeName="IdentityType" InstanceCount="3">
20         <SingletonPartition />
21       </StatelessService>
22     </Service>
23   </DefaultServices>
24 </ApplicationManifest>
```

Program 7.3: A simplified application manifest in XML format describing parts of the allReady identity service; line 19 defines the number of service instances, line 6 (highlighted in blue) includes the service manifest from Program 7.4, lines 3 and 8 (highlighted in orange) show how a parameter is passed through to the included service manifest

configuration parameters into a container, environment variables are used.

The declarations of application type and service type are tightly coupled. To create a new service type it has to be specified in the service manifest. In addition, the service type must be imported into the application. If different teams are responsible for the services within a single application manifest, they have to coordinate updates to the application manifest. Moreover, no clear ownership of the application manifest is given and further problems occur if teams use different source code repositories. Also, if a failure happens during a service upgrade, the whole application upgrade is rolled back and blocks upgrades of other services within the application [Mic19g]. Due to this tight coupling between application and service manifest, it is advisable to declare a microservice on the application level. In other words, an application type is equivalent to a microservices.

Since the application model supports several programming models, its struc-

```
 1 <ServiceManifest Name="IdentityPkg" ...>
 2   <ServiceTypes>
 3     <StatelessServiceType ServiceTypeName="IdentityType" ... />
 4   </ServiceTypes>
 5   <CodePackage Name="Code" ...>
 6     <EntryPoint>
 7       <ContainerHost>
 8         <ImageName>allreadyacr.azurecr.io/allready.identity@sha256:e375d7...
 9         </ImageName>
10       </ContainerHost>
11     </EntryPoint>
12     <EnvironmentVariables>
13       <EnvironmentVariable Name="BasePath" Value="" />
14     </EnvironmentVariables>
15   </CodePackage>
16   <Resources>
17     <Endpoints>
18       <Endpoint Name="IdentityTypeEndpoint" Protocol="http" UriScheme="http"
19                 Port="80" CodePackageRef="Code" />
20     </Endpoints>
21   </Resources>
22 </ServiceManifest>
```

Program 7.4: A simplified service manifest in XML format describing parts of the allReady identity service; line 1 defines the name of service that is referenced by the application manifest in Program 7.3, line 13 (highlighted in orange) shows the definition of an environment variable that is overridden by the application manifest

ture is not ideally aligned for containerized microservices. In order to pass a configuration variable from the build pipeline to the container, the variable must be passed through several layers of the manifests: the service manifest defines an environment variable, the application manifest defines an override for this environment variable, and to set this override, an application parameter is specified. Furthermore, the container image and its repository are specified in the service manifest, while the repository credentials are specified in the application manifest.

7.3.3 Differences between Linux and Windows clusters

Service Fabric supports clusters of Linux or Windows nodes, which only allow to run Linux or Windows containers respectively. A combination of both node types is not possible in a single cluster. If a microservice is implemented with the .NET Framework, the usage of Windows containers is mandatory. Although the .NET Framework is not an optimal solution for a microservice, it might be the case in a lift-and-shift scenario or when porting a service – as done in Subsection 5.3.3 – would be too expensive. Contrary to the .NET Framework, .NET Core doesn't require Windows containers and since allReady is entirely based on .NET Core, it can be deployed to Linux and Windows clusters.

Although Service Fabric supports container orchestration on Linux and Windows, differences in features have to be taken into account [Mic18b]. Linux clusters lack the following features:

- Service Fabric on Linux doesn't provide a reverse proxy out of the box.
- Linux clusters are only supported in the Azure cloud. On-premises installations are not supported.
- The testability subsystem is not available on Linux clusters.
- The EventStore, an additional monitoring tool, is not available on Linux clusters.
- It is not possible to use Reliable Collections within a Docker container in a Linux cluster. Contrary to the official documentation [Mic18a], Reliable Collections work on all Windows clusters regardless of their OS version [McK19].

Disadvantages of Windows clusters are mostly due to the peculiarities of Docker for Windows. On a Windows cluster, the operating system of a node and a container must be compatible. In an optimal case these two are equal; this allows to run containers using process isolation, sharing the host kernel. But Service Fabric can also run containers that are built using an older OS version. This is achieved with Hyper-V isolation where each container runs inside its own virtual machine. In other words, process isolation enables better resource utilization. These circumstances result in the following drawbacks:

- In order to make use of process isolation, every container requires to be rebuilt when the cluster OS is updated.
- This problem is also evident when public container images are used because most of them are not rebuilt for newer Windows versions. As a result, developers cannot use the public image and must create and manage such images themselves.
- Considering the allReady application, some microservices require a Redis database to store state. On Windows this creates a problem because the development of Redis for Windows has been discontinued. The latest version is 3.2.100 and was released in 2016 [Mic16]. An alternative to Redis would be to store state in Reliable Collections

but this would make it difficult to compare the performance with the other orchestration platforms. To ensure equal conditions for the load tests, every platform uses Redis version 3.2. Although this version is not recommended for production use anymore, the type of data storage is not a factor in load testing.

7.4 Cluster state management and scheduling

In a scalable microservices application there are many independent moving parts resulting in a high density of services on a cluster node. Each service consumes resources, like CPU time or memory. To mitigate resource starvation and provide a service with enough resources to function properly, resource governance is required. Its purpose is to limit a microservice' resource consumption and thus isolating it from other microservices co-existing on the same node. Resource governance also improves isolation on a higher logical level: in a multi-tenant cluster it can restrict the resources available to each customer. Another use case could be to run the development and the QA environment in the same cluster but isolate them by limiting their resources.

The consumption of resources is the basic metric used by an auto-scaling mechanism. Available resources on the cluster nodes indicate to the orchestration platform when to provision new nodes or remove existing ones. Moreover, the average consumption of all instances of a microservice indicates when to increase or decrease the amount of instances. Scaling is possible on two axis: vertically by increasing the resources a service is allowed to use, and horizontally by increasing the number of instances. One reason to apply the microservices pattern is to enable horizontal scaling. The smaller a service is, the more efficient horizontal scaling works. This section focuses on scaling an application horizontally.

In order to restrict which services are placed on which nodes, constraints are utilized. These can be static constraints that indicate a service' requirements to a node, like an attached disk. Constraints can also by dynamic, i.e. they depend on other services because they either require to run on the same node or must not be scheduled on the same node.

This section also discusses how to manage state within containerized services. Using the example of a Redis database, it is demonstrated how state can be persisted beyond the lifecycle of a container. The deployment of this Redis database consists of two instances: a master and a replica with both instances continuously persisting their data. The combination of a single master and a single replica is not suited for a production environment but enough to prototypically demonstrate the implementation of stateful microservices.

7.4.1 Azure Kubernetes Service

Kubernetes supports up to 300 000 containers on a cluster with a maximum of 5000 nodes [Kub19b]. AKS further limits this number to a maximum of 800 nodes, each running no more than 250 pods [Mic19b]. In order to dynamically scale the number of nodes in a cluster, Azure Kubernetes Service utilizes the functionality of virtual machine scale sets (VMSS). A virtual machine scale set is a logical group of virtual machines of the same type and provides management capabilities to increase or decrease the number of instances. A VMSS represents a so called node pool.

AKS supports clusters consisting of multiple node pools. This allows to have multiple VMSS with different capacities, which can be used to optimize performance. For example, microservices that have high memory consumption can be scheduled on memory-optimized virtual machines.

Resource governance

Kubernetes uses a fine-grained declarative approach on container level to predefine a microservice' resource requirements. Hereby, a container specifies the minimum amount of CPU time or memory it requires, this is called a resource request. Additionally to these two basic resource types, it is also supported to specify the number of huge pages[4]. If a pod contains more than one container, the container requirements are added up. These metrics are then used by the kube-scheduler to determine on which node the pod will be scheduled. It is also recommended that a container specifies an upper limit for its resource consumption. In case it exceeds this limit, the container might be terminated [Kub19k].

The concept of namespaces within a cluster allows to group microservices, more precisely their API resources, into logical areas, e.g. to run development and QA environments side-by-side, or to separate tenants in a multi-tenant cluster. To mitigate the risk of resource starvation, these namespaces can be restricted in their available resources [Kub19k].

[4]A huge page is a chunk of memory that is larger than the default page size.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: allready-identity
spec:
  maxReplicas: 10
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: allready-identity
  targetCPUUtilizationPercentage: 70
```

Program 7.5: An HPA object in YAML format describing how the deployment object of the allReady identity service (highlighted in blue) is scaled.

Auto-scaling

In a Kubernetes cluster two kinds of scaling mechanisms on pod level are possible: the horizontal pod autoscaler (HPA) and the vertical pod autoscaler (VPA). The VPA adjusts the resource requests and limits of a pod but since AKS only provides support for the HPA, vertical scaling won't be discussed further.

A horizontal pod autoscaler is bound to a replication controller – for example, in Program 7.5 this replication controller is a deployment object. The HPA periodically checks the CPU utilization of all targeted pods within the replication controller (the default time interval is 15 seconds). Based on the targeted utilization, it calculates how many instances are required to optimally achieve this target. If the calculated instance count differs from the current instance count, the HPA instructs the deployment to increase or decrease the number of pod replicas. The new beta version of the HPA also allows to scale based on memory or custom metrics [Kub19g].

Storage

Oftentimes it is inevitable to require state within a microservices. In regards to the allReady application, examples are the Redis databases that are used by the Identity and MVC services. In order to not lose the entire data when a Redis pod is rescheduled, Redis provides the persistence feature. This allows to store the data on disk. When running inside a container this requires the orchestration platform to mount the Docker volume to the underlying cloud

```
 1 apiVersion: apps/v1
 2 kind: StatefulSet
 3 spec:
 4   replicas: 2
 5   template:
 6     spec:
 7       containers:
 8       - name: redis
 9         image: redis:3.2
10         volumeMounts:
11         - name: data
12           mountPath: /data
13   volumeClaimTemplates:
14   - metadata:
15       name: data
16     spec:
17       accessModes: [ "ReadWriteOnce" ]
18       resources:
19         requests:
20           storage: "10Gi"
```

Program 7.6: A stateful set object describing the replicated Redis setup used by the allReady MVC service; line 19 sets the access mode to `ReadWriteOnce` which means only a single pod replica can access the persisted volume (mandatory properties `.spec.template.metadata.labels` and `.spec.selector.matchLabels` removed for the sake of brevity)

infrastructure [Kub19l].

In Kubernetes this is accomplished by using a persistent volume that maps to any kind of cloud storage. AKS allows to dynamically provision persistent volumes by creating so called persistent volume claims (PVC). These instruct the cloud provider to provision storage according to the specification in the PVC. Persistent volume claims and persistent volumes form a 1-to-1 relationship.

In order to operate Redis in a highly available and fault-tolerant way, multiple Redis instances are deployed that synchronize their state; to simplify this, a stateful set is used. A stateful set is a Kubernetes object that, similar to a deployment object, allows to specify the number of pod replicas that must be running in the cluster. Other than a deployment, it also provides a declarative approach to create a PVC for each pod and attach the provisioned storage to the container. The storage can be configured to remain available when a pod is terminated. Thus, it can be attached again when the pod is rescheduled. Program 7.6 shows a simplified declaration of a stateful set within the MVC microservice.

```
 1 apiVersion: apps/v1
 2 kind: StatefulSet
 3 spec:
 4   replicas: 2
 5   template:
 6     metadata:
 7       labels:
 8         app: mvc-redis-ha
 9     spec:
10       affinity:
11         podAntiAffinity:
12           requiredDuringSchedulingIgnoredDuringExecution:
13             - labelSelector:
14                 matchLabels:
15                   app: mvc-redis-ha
16               topologyKey: kubernetes.io/hostname
```

Program 7.7: Part of a stateful set object declaration specifying an anti-affinity between its pod replicas. The self-reference is accomplished by the label selector `app=mvc-redis-ha` in line 15 matching its own label in line 8 (removed mandatory properties `.spec.template.metadata.labels` and `.spec.selector.matchLabels` for the sake of brevity)

Constraints and affinity

In a Kubernetes cluster, arbitrary labels can be assigned to nodes, e.g. the disk type attached to a node. By using the node selector property, a pod can define on which nodes it is allow to run. For example, it can require the node to have an SSD attached. These node selectors are hard requirements. Furthermore, a pod can also specify an affinity (or anti-affinity if an aversion is required) to other pods. This can either be a hard or a soft requirement, where a soft requirement indicates a preference but does not need to be strictly obeyed by the kube-scheduler [Kub19a].

By incorporating anti-affinity between the Redis pods into the stateful set, it is achieved that only one Redis replica is scheduled on a single node. This is important in order to improve fault tolerance. Program 7.7 demonstrates how this anti-affinity between replicas of the same pod definition is specified.

7.4.2 Service Fabric

Service Fabric does not define an upper limit as to how many containers can be orchestrated or how many nodes a cluster can contain. Ramaswamy

[Ram17] demonstrated that Service Fabric is capable of scheduling over one million containers.

Just like AKS, Service Fabric utilizes virtual machine scale sets to provision and manage nodes in the cluster. Service Fabric also supports clusters consisting of multiple node pools so that microservices can be scheduled on virtual machines that optimally support their resource requirements.

Resource governance

In comparison to AKS, Service Fabric has a very similar strategy on how to handle resource governance. The limits of CPU cores and memory are specified on the service level. These metrics are also used to determine on which node a new service is scheduled. Additionally, container resources can be limited by a wide variety of metrics: swap memory, memory reservation, maximum IO rate (operations per second) for read and write, maximum IO bandwidth (in bytes), block IO weight, disk quota, kernel memory (on Linux) or the size of `/dev/shm` (on Linux) [Micl].

Other than Kubernetes, Service Fabric does not implement the concept of namespaces. This complicates the resource governance for multiple tenants. Even though separate application instances can be created for each tenant, constraining all the moving parts of a microservices application individually is cumbersome and hard to maintain.

Auto-scaling

Service Fabric allows an application to scale its service instances by defining a lower and upper threshold for a metric. In a specified interval, it checks the metric by considering the average value from all instances. In case this average lies outside the thresholds, it decreases or increases the instance count by a specified number. When the instance count is changed, Service Fabric checks the thresholds again after the next interval has elapsed [Mic18d]. If the added instances are still not enough to handle the workload, failures might occur.

Service Fabric limits the maximum number of instances that can be created for a service. This limit corresponds to the number of nodes within a cluster. In other words, at most one instance of a service can run on a particular node [Mic19j]. Although it would be possible to create another identical service with multiple instances and therefore double the instance count, it is not possible to assign the same DNS name to both identical services. Since for the load tests the allReady application is only deployed on a three node cluster, Service Fabric's auto-scaling mechanism would only scale a microservice between one and three instances. Furthermore, an instance count of one would not ensure fault-tolerance. Therefore, the auto-scaling mechanism is not implemented due to the lack of benefit.

Concerning resource governance, the restriction of one service instance per node can also become a problem on small clusters: restricting a service' available resources too much can rapidly increase the need for more nodes, resulting in higher financial costs.

Storage

To enable Redis to persist its data, the docker volume has to be mounted. Service Fabric provides a so called Azure Files volume driver that attaches an Azure Files[5] storage to the Docker volume. This volume driver is built atop of a Docker volume plugin. But this Docker volume plugin does not support a high rate of IO operations and would therefore introduce high latency [Doc18]. Moreover, Service Fabric does not provide a volume driver to attach an Azure Disk, which is suited for high IO rates, to a Docker volume [Jan19]. As a result, a manual approach is required to achieve persistence on a disk.

```
<Volume
    Source="/datadisks/disk1/appdata/MVC.SFType/MVC.RedisReplica"
    Destination="/data" />
```

Program 7.8: Attaching the Docker volume `/data` to a directory in the virtual machine's data disk. `/datadisks/disk1` points to the disk, `/appdata` is a sub directory defined by convention, `MVC.SFType` and `MVC.RedisReplica` are the names of the application and the service respectively

A VMSS allows to specify a configuration for a data disk. To each virtual machine in the scale set, a data disk will be attached according to the configuration. After formatting the data disk, it can be attached to the Docker volume of the Redis container. Program 7.8 shows how the data disk is attached to the Docker volume of a Redis replica. A downside of this method is that a disk is bound to the virtual machine and not to the container; therefore, every virtual machine requires a disk to run the container. Another disadvantage is that every service running on the virtual machine has

[5]https://docs.microsoft.com/en-us/azure/storage/files/storage-files-introduction

```
 1 <SetupEntryPoint>
 2   <ExeHost>
 3     <Program>selector.cmd</Program>
 4     <Arguments>MVC.SFType/MVC.RedisReplica</Arguments>
 5     <WorkingFolder>CodePackage</WorkingFolder> <!-- script location -->
 6   </ExeHost>
 7 </SetupEntryPoint>
 8 <EntryPoint>
 9   <ContainerHost>
10     <ImageName>allreadyacr.azurecr.io/allready.mvc.sfwin.redis</ImageName>
11     <Commands>--slaveof,allready-mvc-redis,7379</Commands>
12   </ContainerHost>
13 </EntryPoint>
```

Program 7.9: Definition of the service manifest entrypoints to deploy a Redis replica

access to this data. In order to not interfere with other services following this approach, each service creates it's own directory. All developers are required to follow this convention but since it is not possible to enforce this convention, problems might occur.

The directory is created using an entrypoint script. Program 7.9 shows two entrypoints in the service manifest with the following characteristics:

- Line 3 specifies the `selector.cmd` script that is run before starting the container.
- The argument on line 4 passes the directory to be created by the script.
- Line 10 defines the custom-built Redis image for Windows; on a Linux cluster, the public Docker image `redis:3.2` is used.
- Line 11 shows the command that is passed to the docker container: here the container is deployed as a Redis replica. This replica instance connects to the Redis master using the master's DNS name `allready-mvc-redis`, and keeps their states synchronized. The service manifest of the Redis master looks similar. It is important to not use any whitespace characters within the XML tag since this would cause Service Fabric to pass the arguments incorrectly to the Docker container.
- Remark: `--slaveof` is used due to the older Redis version; in newer Redis versions this flag is named `--replicaof`.

Because higher privileges are required to create directories, a dedicated setup entrypoint executes the script as Service Fabric administrator. Depending on whether the cluster consists of Linux or Windows nodes, either a bash script (Program 7.10) or a batch script (Program 7.11) is executed. The `selector.cmd` script (Program 7.12) delegates the setup to the correct script depending on the operating system it is executed on. Service Fabric requires the explicit declaration of the `#!/bin/bash` shebang; if not defined, the script will fail because the default shell `/bin/sh` is not installed on a Service Fabric node.

```
#!/bin/bash
dir="$1"
volume="/datadisks/disk1"
absPath="$volume/appdata/dir"
sudo mkdir -p "$absPath"
```

Program 7.10: setup.bat

```
set dir=%1
set vol=F:
if not exist %vol%\appdata\%dir% (
    mkdir %vol%\appdata\%dir%
)
```

Program 7.11: setup.sh

```
1 #!/bin/bash
2 :<<BATCH
3     @echo off
4     echo Running setup.bat
5     call setup.bat %*
6     exit /b
7 BATCH
8     echo "Running setup.sh"
9     sudo ./setup.sh "$@"
```

Program 7.12: The `selector.cmd` script to run the correct setup script depending on the host OS: on Linux the here-document instruction (<<) reads multiple lines into stdin until the `BATCH` mark on line 7 is reached but does not interpret this batch code; on Windows line 2 is ignored due to the leading colon, thus the script starts at line 3 and runs until it explicitly exits at line 6; this file must use Linux line endings

Constraints and affinity

Just like Kubernetes, Service Fabric also allows to add static tags to a node. These tags are called node properties and can be used in combination with placement constraints to restrict the scheduling of services.

By declaring affinity, services can be placed on the same nodes. Other than

Kubernetes, Service Fabric does not support a concept to specify an aversion or anti-affinity between services. This means there is no guarantee that all Redis instances (master and replica) are located on different virtual machines. In case all instances are scheduled on the same host, a failure in this virtual machine would cause the entire Redis database to become unavailable.

Another problem is that the provisioned disks are not bound to the service instance but rather to the virtual machine. In case a Redis replica is rescheduled to another node, the newly created container does not have access to the data anymore. This issue is mitigated by Redis internal replication mechanism, which allows the container to recreate the data. This is not an optimal solution. Service Fabric allows to declare a so called move cost which is considered by the scheduling algorithm. To reduce the probability of rescheduling to another node, the move cost of all Redis instances is set to the value *high*. This instructs Service Fabric to not move the container except it is inevitable. It is not possible to declare a service as unmovable [Mic17h].

7.5 Providing high availability and fault tolerance

The cluster internals on how each orchestration platform achieves high availability and fault tolerance have already been discussed in the Sections 7.2 and 7.3. Therefore, this section focuses on a developer's perspective and explains measures they can take to improve fault tolerance.

The key concept is adding redundancy to all microservices. As already discussed, both orchestration platforms provide mechanisms for the developer to declare the desired number of service instances, and can also adapt this number automatically depending on the workload.

When provisioning a cluster with Azure Kubernetes Service or Service Fabric, it is recommended to handle incoming traffic using a public load balancer. Independent of the orchestration platform an Azure Load Balancer[6] is created automatically in front of the cluster. The load balancer dispatches incoming requests to the virtual machines in the cluster. To ensure the load balancer only routes traffic to available virtual machines, it uses health probes. If a virtual machine doesn't respond to a probe, the load balancer marks it as temporarily unavailable.

[6]https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview

A similar approach to ensure the availability of particular services is to utilize health checks. These are implemented by providing a service endpoint that is used by the orchestration platform to verify its state. How a health check endpoint is implemented is up to the developer. It is important not to include dependencies, e.g. a working database connection or other microservices, into these checks. This can cause cascading failures throughout the application, for instance, when there are network problems or the database is temporarily unavailable. Instead of removing services from the application completely, graceful degradation is recommended to maintain functionality which is independent of the unavailable dependency.

In a continuous-development environment, developers deploy updates to their services at a very high frequency. Such a deployment process must not cause downtime. A high rate of daily deployments, involves a higher risk of failures during the update process. Therefore, deployment errors are to be expected and a recovery process must be in place to bring the application back into a functioning state.

7.5.1 Azure Kubernetes Service

The following aspects explain how Azure Kubernetes Service is ensuring high availability and mitigates the impact of failing services.

Health checks

Kubernetes uses probes to check if a pod is functioning correctly. Containers within a pod provide endpoints, for example via HTTP, that are called periodically by the kubelet. To signal the service is healthy, an endpoint returns a success response code. There are two different kinds of probes that can be provided by a container: readiness and liveness probes. The readiness probe indicates if a container is ready to receive requests. If the probe fails, Kubernetes stops forwarding requests to this instance until the readiness probe succeeds again. This probe is also helpful if a container's startup procedure is time-consuming: after completing the startup, it can signal its ready-state in order to receive requests. The liveness probe is used to determine a faulted container state. Per default, a failing liveness probe results in Kubernetes terminating and restarting the container. This should only be used if the failure cannot be resolved within the service. For example, this might be the best option if a deadlock has occurred [Kub19m]. The kubelet must not start checking the liveness probe before the container is initialized. Otherwise the

```
readinessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 10
  failureThreshold: 3
livenessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 20
  failureThreshold: 10
```

Program 7.13: Readiness and liveness probes within the container specification of a deployment object; `initialDelaySeconds` specifies when the kubelet starts checking the probe after startup, `failureThreshold` specifies the number of consecutive failures until actions are taken

container will always be restarted before it is available, causing an endless initialization loop. The property `initialDelaySeconds` specifies how long the kubelet waits before checking a probe. Dinesh [Din18] recommends using the 99th percentile initialization time.

Program 7.13 shows a containers declaration of both probes in form of HTTP requests. It is possible to point both probes to the same endpoint. In this case, the recovery actions for the liveness probe should be triggered later than the actions for the readiness probe: in the event of an overload, a service has time to recover instead of being terminated immediately.

Updating services

Kubernetes supports two different strategies to update a deployment object: recreate and rolling update. The recreate strategy deletes all existing pods before it creates pods of the new version. Here the deployment is temporarily unavailable and only useful during development. The rolling update strategy alternately creates new and deletes old pods, keeping the total number of pods within a relative deviation of 20 percent. The upper and lower limit of this deviation can be specified explicitly using the `maxSurge` and `maxUnavailable` properties respectively [Kub19f].

Additionally, a rollback mechanism is required to recover from update failures and mitigate the risk of downtime. Kubernetes does not provide a declarative approach to automatically rollback on deployment failures but requires the developer to manually rollback using the command line interface. To enable automatic rollback, the Helm package manager – explained in Subsection 7.9 – can be utilized. More complex strategies, like blue-green or canary deployments, are also covered Subsection 7.9.1.

Further measures

As discussed in Section 7.4, namespaces are a concept to limit resources for different parts of the application or cluster. Similar to the bulkheads pattern from Subsection 4.4.2, namespaces ensure fault tolerance by prohibiting microservices to over-commit resources which would cause starvation of other microservices.

7.5.2 Service Fabric

The following measures describe how Service Fabric is achieving high availability and fault tolerance. These measures are compared with those of AKS.

Health checks

Service Fabric uses a hierarchical model to represent a cluster's health state where each child entity reports its health state to its parent. In regards to containerized microservices, simplistically speaking, the container reports its state to the service instance which it is encapsulated by. The service instance then reports to its application instance which in turns reports to the cluster. Depending on the configured health policy, an error in the child entity results in an error, a warning or is ignored [Mic18e].

In order to terminate a container that is in an unhealthy state, Service Fabric incorporates Docker health checks: in the Dockerfile a `HEALTHCHECK` command is defined that is executed at a regular interval of time. For example, this can be a HTTP request to a health endpoint provided by the microservice. If this health check fails, Service Fabric marks the container with a warning. By default, Service Fabric does not terminate the container. In order to automatically restart it, the containers' health configuration can be modified. Program 7.14 shows how this configuration is declared. The combination of Docker health checks with the restart configuration enables the same functionality as a Kubernetes liveness probe.

```
1 <ContainerHostPolicies CodePackageRef="Code">
2   <HealthConfig
3     IncludeDockerHealthStatusInSystemHealthReport="true"
4     RestartContainerOnUnhealthyDockerHealthStatus="true" />
5 </ContainerHostPolicies>
```

Program 7.14: A container's health configuration; line 3 instructs Service Fabric to restart the container when the health check fails

```
<StatelessServiceType ServiceTypeName="MVCType" ...>
  <Extensions>
    <Extension Name="Traefik">
      <Labels xmlns="http://schemas.microsoft.com/2015/03/fabact-no-schema">
        <Label Key="traefik.enable">true</Label>
        <Label Key="traefik.backend.healthcheck.path">/health</Label>
        ...
```

Program 7.15: The manifest of the MVC service registering a health endpoint for Traefik

Service Fabric does not provide an equivalent mechanism to a Kubernetes readiness probe. In order to schedule ingress traffic only to operational containers, Traefik is deployed as an internal load balancer. Each service registers with Traefik and provides a health check endpoint that is periodically checked by Traefik [Con19]. Only if this endpoint returns a success status code, Traefik will forward requests to the service instance – see Program 7.15 for an example configuration. Section 7.8.2 covers Traefik in more detail.

Updating services

A Service Fabric cluster is divided into multiple update domains. In a rolling upgrade, Service Fabric applies the upgrade to one update domain at a time. When an update domain is considered healthy after applying the changes, the upgrade proceeds with the next update domain. An upgrade is only successful when all update domains are healthy. In case an upgrade fails, Service Fabric supports to roll back to the initial version automatically [Mic18g].

It is also possible to upgrade all instances at once. Since this procedure deletes the old instances first and then creates new ones, this causes downtime. For this reason, upgrading all instances at once is only recommended for development purposes [Mic18g].

Further measures

Since Service Fabric has no concept of namespaces, resource limitations must be specified at a service or container level. This is error prone and does not allow a cluster administrator to enforce resource limits.

7.6 Ensuring security

Running containers should follow the principle of least privilege. This means a service or container only has access to resources that it requires to accomplish its responsibilities. Per default, a Docker container is run with user ID (UID) 0 and thus has root access. Obviously, this does not follow the principle of least privilege and allows a potential attacker to further exploit the compromised container and possibly gather information about the network, orchestration system or other services. To mitigate this vulnerability, it is recommended to apply the `USER` instruction in the Dockerfile. On Linux for instance, `USER 1000` sets the user ID to 1000 which does not have root access. It is not required that a user with this id already exists. In a Windows container, a user has to be created before it can be specified with the `USER` instruction [Docc].

A disadvantage of this method is its susceptibility to developer errors because the instruction must be specified explicitly in every Dockerfile. It also hinders the direct usage of public images that don't apply this instruction. This section evaluates the capabilities of each orchestration platform to restrict root access within containers on a higher and more general level.

Furthermore, this section covers how to manage credentials and other sensitive information in the cluster.

7.6.1 Azure Kubernetes Service

The following aspects explain the most crucial security features a developer has available on Azure Kubernetes Service.

```yaml
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: unprivileged-only
spec:
  privileged: false
  allowPrivilegeEscalation: false
  runAsUser:
    rule: MustRunAsNonRoot
```

Program 7.16: A pod security policy to disallow root access for containers; not available for production in AKS

Restricting root privileges

Kubernetes allows to set a security context for pods and their containers. The security context declares a user ID that executes all container processes. This declaration takes precedence over the specified UID in the Docker image. Using a security context allows to run public container images without root privileges even if they do not use the `USER` instruction [Kub19e].

Still, a security context has to be specified in the pod definition which can cause vulnerabilities if developers don't explicitly set it. In order to disable root access for containers for the entire cluster, Kubernetes implements the concept of pod security policies [Kub19o]. A pod security policy consists of several rules that are checked when a pod is created. Program 7.16 declares a policy that forces all containers to run without root privileges and hinders a container from escalating its privileges. For AKS clusters this feature is in preview state and not ready for production use [Mic19i].

Secret management

A pod references secrets that it needs to use in its YAML declaration. A secret is represented by a Kubernetes object. Secrets managed by Kubernetes are encoded in base64; this makes it possible to also store certificates that are originally in binary format. As a protective measure, a secret is only written to temporary memory on nodes, where pods reference the secret. If all these pods are removed from the node, the secret is removed, too. A pod can consume the secret in two ways: either the secret is injected via an environment variable or mounted as a volume [Kub19p].

Furthermore, Microsoft is working on the integration of Azure Key Vault

```
<Principals>
  <Users>
    <User Name="SetupLocalSystem" AccountType="LocalSystem" />
  </Users>
</Principals>

<ServiceManifestImport>
  <ServiceManifestRef ServiceManifestName="MVC.RedisPkg" ... />
  <Policies>
    <RunAsPolicy CodePackageRef="Code" UserRef="SetupLocalSystem"
      EntryPointType="Setup" />
...
```

Program 7.17: Application manifest showing the configuration and referencing of a privileged system account principal

into AKS: this allows to manage sensitive data in Azure Key Vault and mount it as a volume to pods [Mic19a].

7.6.2 Service Fabric

Service Fabric has similar capabilities as Azure Kubernetes Services with regard to secret management. For containers, however, Service Fabric only offers limited security features.

Restricting root privileges

Service entrypoints on Service Fabric are executed with an unprivileged account. In order to run setup scripts, as done in Section 7.9, higher privileges are required. These are obtained through the definition of a user principal for the local system account. On Windows this is the `LocalSystem` account, on Linux this maps to root (UID 0). This principal is referenced in a `RunAsPolicy` for the entrypoint [Mic18f]. Program 7.17 demonstrates how this principal is passed to the setup entrypoint of the MVC service' Redis container.

The privilege restriction is only implemented for native processes, i.e. script execution or executables running directly on Service Fabric, but not for Docker containers: Service Fabric controls the Docker daemon (dockerd) and instructs it to run a container image. The Docker daemon then delegates this to its default container runtime known as runC. Service Fabric executes dockerd with elevated privileges; containers are also run with elevated privileges if not specified explicitly with the `USER` instruction in the Dockerfile. Service Fabric allows to specify the `ContainerServiceArguments` parameter which is used to start dockerd with custom arguments. Also, dockerd provides a mechanism to configure its runtime with custom arguments. To apply custom arguments to the runtime, the daemon configuration file must be used; the command line interfaces does not support it [Docb]. Since the specified `ContainerServiceArguments` are only applied to dockerd's command line arguments, it is not possible to pass an unprivileged user id from the Service Fabric configuration through to the container instance. To summarize, Service Fabric does not provide assistance for the developers to follow the principle of least privilege with regard to containers.

With Hyper-V isolation, Service Fabric on Windows supports a higher level of isolation by hosting each container on is own virtual machine and thus providing each container its own kernel.

Secret management

Service Fabric provides the Central Secret Store (CSS) to safely handle secrets within the cluster. The CSS is a replicated cache store that holds sensitive data in memory. This data is stored encrypted; for decryption the private key of the encryption certificate is required. A secret can either be declared using an ARM[7] template or created imperatively via a REST API. To make a secret available to a container, it is declared in the service manifest. Same as AKS, in order to use the secret, the developer can choose between environment variable injection and volume mounting [Mic19k].

Beginning with Service Fabric version 7, the CSS supports a preview feature that incorporates Azure Key Vault. With this feature, the secrets are stored in Azure Key Vault and the CSS acts as a caching layer [Mic19e; Sen19].

7.7 Simplifying networking

This section focuses on the fundamental networking that enables communication between containers on different nodes. Higher-level mechanisms that facilitate communication without knowing a service' IP address are not within the scope of this section; these mechanisms are explained in Section 7.8.

[7]Azure Resource Manager (ARM) is a declarative approach to specify cloud resources

```
spec:
  containers:
  - name: allready-identity
    image: allreadyacr.azurecr.io/allready.identity@sha256:e375d7...
    ports:
    - containerPort: 80
```

Program 7.18: Pod template specification of a container listening on port 80

7.7.1 Azure Kubernetes Service

In a Kubernetes cluster all pods can communicate with each other, independent of which physical node they are running on. The Kubernetes network model assigns every pod a unique cluster-internal IP address. This removes risk of conflicting ports on cluster nodes, which would potentially require port coordination among development teams [Kub19c]. Program 7.18 shows how a pod's port 80 is opened to receive traffic.

7.7.2 Service Fabric

Service Fabric supports two networking modes: nat, which is the default, and open. Using nat mode requires port coordination between all microservices because multiple containers listening on the same port will result in failures. If different teams are in charge of different microservices, this entails considerable effort and is therefore not feasible. Using open mode, every service has a dynamically assigned IP address, eliminating the need for port coordination mentioned above. Program 7.19 explains how a container's port is made available to other services within the cluster. A restriction on Linux clusters is that an application cannot consist of services with different networking modes. Using open mode requires to manually add a network interface (NIC) to the VMSS for each assignable IP address. This means the maximum number of IP addresses in the cluster is equal to the number of network interfaces multiplied by the number of nodes. A maximum of 50 IP addresses per node has been tested by Microsoft [Mic18h].

It is not possible to force the networking mode on cluster-level, i.e. in order to avoid port collisions, each service has to specify the open mode when it requires to listen on a static port.

```
 1 <CodePackage Name="Code" ...>
 2   <EntryPoint>
 3     <ContainerHost>
 4       <ImageName>allreadyacr.azurecr.io/allready.identity@sha256:e375d7...
 5       </ImageName>
 6     </ContainerHost>
 7   </EntryPoint>
 8 </CodePackage>
 9 <Resources>
10   <Endpoints>
11     <Endpoint
12       Name="IdentityTypeEndpoint" Protocol="http" UriScheme="http"
13       Port="80" CodePackageRef="Code" />
14   </Endpoints>
15 </Resources>
```

Program 7.19: Specification of a container listening on port 80 within a service manifest; line 13 publishes port 80 for the `allready.identity` container which is referenced via the code package name (highlighted in blue)

7.8 Enabling service discovery

Owing to frequently changing service locations, an automatic approach is required to enable service communication. Communication can be broken down into two categories: service to service, and client to service. The former category is mainly about how services can provide endpoints that are easily discoverable and accessible. Both platforms provide a DNS server for a server-based discovery approach. The latter covers how services within the cluster can be exposed to external clients in a secure way, and how to handle the resulting, incoming traffic. An integral pattern is the API gateway, which is explained in Subsection 4.4.4. The Azure Load Balancer residing in front of the cluster, operates on layer 4 and thus is not able to apply a HTTP-based routing. Layer 7 routing has to be implemented by the orchestration platform.

7.8.1 Azure Kubernetes Service

Pods in a Kubernetes cluster are ephemeral, i.e. they are created and terminated with a high frequency. This results in constantly changing IP addresses. Kubernetes has several concepts that simplify service discovery in the cluster.

Service discovery

As explained in Section 7.2, Kubernetes provides a service object – in this Subsection from here on referred to just as *service* – that keeps track of a set of pods. This service also has a unique cluster-internal IP address and can be consumed by other pods in two different ways: via an environment variable that injects the IP address of the service into a container, or by utilizing the internal DNS server. The first approach is not viable because to inject the IP address of a service into the container, the service must already exist when the container is created. The service' IP address can be discovered by resolving its name – additionally with its namespace if required – using the DNS server. A service can load balance requests across multiple deployments. This can be utilized when deploying two versions of a microservice side-by-side, each as its own deployment. Requests to the service' DNS name are load balanced between both versions but the existence of multiple deployments is unapparent from the caller's perspective. A service also allows to map a container's port to a custom one. For example, when a container is listening on port 8080, the service can receive HTTP requests on port 80 and redirect them to the container's port 8080.

Exposing services

Although it is possible to directly expose a service outside the cluster via a dedicated Azure Load Balancer, it is recommended to provide a single entrypoint to external clients that acts as a facade for the internal services. To achieve this, Kubernetes offers an ingress object [Kub19h]: An ingress is comprised of a standardized resource definition and an implementation-specific ingress controller. The ingress resource is specified by the developer and interpreted by the ingress controller. When an ingress object is created, an Azure Load Balancer is provisioned that forwards external layer 4 traffic to the ingress controller. An ingress controller is a reverse proxy operating on layer 7 traffic. There exist several implementations for ingress controllers that all adhere to the ingress resource definition. The available implementations show differences in their functionality but most of them offer a common feature set:

- An ingress provides TLS termination for incoming requests that are then forwarded to back end services via HTTP. This removes the decryption overhead from the back end services.
- It enables more precise load balancing than a service object. A service object uses kube-proxy to route traffic on layer 4. Most implementa-

tions of ingress controllers don't just forward requests to a service, which then does the load balancing, but communicate with the kube-apiserver to keep track of the moving pods. Because ingress controllers operate on layer 7, they can perform more application-oriented load balancing algorithms.

- Using an ingress, several services can be consolidated and made available at the same IP address. Routing rules allow to address different services depending on the host or path.
- An ingress controller can be replicated to ensure high availability.

To expose allReady microservices to external clients, an nginx ingress controller is used – mainly for the reason that it's officially supported by Kubernetes [Kub19i]. All externally accessible services are exposed under the same host name `allready.westeurope.cloudapp.azure.com` and are reachable via different sub paths. This ingress is the single entrypoint for external clients. The ingress resources in Program 7.20 defines how the internal microservices are reachable, e.g. all requests to `/identity` are routed to the identity microservice. As explained in Subsection 6.2.2, the identity microservice is aware of the sub path.

The ingress controller is also responsible to terminate incoming TLS connections. With the integration of cert-manager[8], a Kubernetes tool to manage certificates, automatic issuance and renewal of certificates using Let's Encrypt is possible. Identity Server's authorization code flow, based on OpenID Connection, utilizes HTTP with redirects between the MVC and identity microservice. To keep these redirects from failing, the proxy buffer size of the underlying nginx must be increased. The nginx ingress allows to configure the proxy buffer size with custom `nginx.ingress.kubernetes.io` annotations in the ingress resource. These annotations are specific to the nginx ingress controller. Program 7.20 shows the complete specification of the ingress resource used for the allReady application.

7.8.2 Service Fabric

Service Fabric comes with several limitations on how containers can be addressed within the network. These limitations entail greater effort for the development teams to operate their services without interfering with each other. There are a number of mechanisms, which Kubernetes provides out of the box, that have to be implemented manually in a Service Fabric cluster. Therefore, these aspects are explained in greater detail.

[8]https://cert-manager.io/docs/

```
 1 apiVersion: networking.k8s.io/v1beta1
 2 kind: Ingress
 3 metadata:
 4   name: allready-ingress
 5   annotations:
 6     kubernetes.io/ingress.class: nginx # controller implementation
 7     certmanager.k8s.io/cluster-issuer: letsencrypt-prod
 8     nginx.ingress.kubernetes.io/proxy-buffering: "on"
 9     nginx.ingress.kubernetes.io/proxy-buffer-size: 8k # default=4k
10 spec:
11   tls:
12   - hosts:
13     - allready.westeurope.cloudapp.azure.com
14     secretName: acme-crt-secret
15   rules:
16   - host: allready.westeurope.cloudapp.azure.com
17     http:
18       paths:
19       - path: /
20         backend:
21           serviceName: allready-mvc
22           servicePort: 80
23       - path: /main
24         backend:
25           serviceName: allready-main
26           servicePort: 80
27       - path: /identity
28         backend:
29           serviceName: allready-identity
30           servicePort: 80
```

Program 7.20: A complete ingress specification; line 6 defines the ingress controller to use as a back end for this ingress resource, line 7 integrates the certificate issuer object of cert-manager into the ingress, lines 8 and 9 apply custom configuration to the nginx ingress controller, lines 16 to 30 specify the routing rules for incoming requests

Service discovery

Service Fabric runs a highly available naming service that was originally built to be used by its platform-integrated programming model. Each deployed service type is called a named service. It has a unique name consisting of its application and service name. The name is determined by the naming service and cannot be set manually. All instances of a named service share the same name. The naming service keeps track of the network locations of the instances and resolves the name to an IP address. To allow Linux and Windows containers to resolve the network location of other services without being dependent on the platform framework, a cluster-internal DNS server operates on the basis of the naming service, i.e. the DNS server uses the naming service to resolve the IP address of a service endpoint. For containerized services, a service endpoint is equivalent to an exposed container port [Mic17c].

Although it is not recommended to integrate microservices based on a synchronous communication pattern, synchronous HTTP calls still can be required for specific use cases, like the MVC service querying the main service. The MVC service is a special kind of client that resides within the application. To integrate these services, the main service – as well as any other service that provides an API to the MVC service – has to expose its endpoint at port 80. The reason for this is that the A record[9] returned by the DNS service only contains the IP address but not the port number. Although it is possible to retrieve the port number of a service via an SRV lookup, most HTTP libraries do not support this. Therefore, the default port 80 is the only solution without hard-coding ports into consuming services or introducing further discovery mechanisms. This is still a problem for non-HTTP endpoints like Redis: these ports have to be coordinated between the service and the consuming client. Of course using port 80 is also possible. However, this might lead to a scarcity of IP addresses due to the soft limit of 50 NICs per node.

As opposed to Kubernetes, in Service Fabric the DNS name of a service is a direct mapping to a set of identical container instances. This means, there is no ready-to-use mechanism to deploy multiple service versions sharing one common DNS name. As a result, the executing of more complex deployment approaches becomes more difficult.

Exposing services

Service Fabric comes with a built-in reverse proxy to provide services to external clients. The foundation of the reverse proxy is the naming service, which is used to resolve public URIs to internal names. The reverse proxy has the following downsides that make it unsuitable as a publicly accessible API gateway [Mic17d]:

- It forces a specific URI format for all services that is derived from the naming service. The URI is constructed using the application and service name. Thus, the reverse proxy cannot be used as a facade

[9]An *A record* is a type of DNS record and that maps a domain name to an IP address.

and exposes information about the internal application architecture to external clients by its naming convention.

- Utilizing the built-in reverse proxy results in exposing all internal services to the public. This poses a security risk as there is no possibility to control which services are accessible through the reverse proxy.
- If a service cannot be reached by the reverse proxy, which is very likely due to services moving location in the cluster, it receives a 404 error response. The reverse proxy then automatically retries the requests to another service instance. These automatic retries cause issues for 404 responses that are returned intentionally by the business logic of a service. Reverse proxy will also retry these requests unless it is explicitly instructed by the service not to by setting the HTTP header `X-ServiceFabric` to `ResourceNotFound`. This couples service logic with specifics of the orchestration platform and is thus undesirable.
- The reverse proxy is only available on Windows clusters.

There are several other layer 7 options available to implement the API gateway pattern and expose microservices in a controlled manner. The following itemization explains them and gives some indication on their applicability:

- **Azure API Management (APIM)**: This Azure product provides a direct integration into Service Fabric. It is deployed in the same virtual network as the Service Fabric cluster and utilizes the naming service to resolve services. Due to the mandatory integration into the same virtual network, the Premium tier is the only available option for production environments [Mic17g]. Since this tier incurs costs of about €2357 per month, APIM is not considered for this prototype [Mica].
- **Azure Application Gateway (AAG):** AAG is another product managed by Azure. Opposed to APIM, AAG cannot be integrated directly into a Service Fabric cluster. AAG is suitable to route requests to different node pools depending on the path. Since the allReady application is running on a single node pool, this routing mechanism is not applicable. AAG is not able to route requests directly to the moving service endpoints because it has no access to the naming service and thus has no information about the endpoint locations [Mic19m].
- **Traefik:** Traefik is an open source reverse proxy that, in its version 1.7, provides an integration with Service Fabric. Other than APIM and AAG, Traefik is not managed by Azure but deployed to the cluster as a separate service. This has the advantage of a self-contained cluster that does not rely on external vendor-specific components. In a prototypical approach, the allReady application was implemented for both Service Fabric operating systems utilizing Traefik as a reverse proxy.

```
<StatelessServiceType ServiceTypeName="MainType" ...>
  <Extensions>
    <Extension Name="Traefik">
      <Labels xmlns="http://schemas.microsoft.com/2015/03/fabact-no-schema">
        <Label Key="traefik.enable">true</Label>
        <Label Key="traefik.frontend.rule.default">PathPrefix: /main</Label>
        <Label Key="traefik.frontend.passHostHeader">true</Label>
...
```

Program 7.21: Registration of the main service that instructs Traefik to forward the `/main` route to the main service

Traefik is available for several orchestration platforms. For each supported platform, Traefik uses a dedicated provider implementation. A provider is a component within Traefik that observes the cluster state using the management API of the specific platform. In the case of Service Fabric, the provider sends periodic requests to the Service Fabric Client REST API[10]. In order to gain access to the REST API, the provider authorizes itself using a client certificate issued by Service Fabric. Since the provider only queries the API, it is recommended to issue a read-only certificate without administrative permissions. When the provider detects that services have changed location or become unhealthy, Traefik is notified by the provider and updates its routing configuration [CG18].

Unlike a Kubernetes ingress, which holds routing configuration in a centralized resource, Traefik on Service Fabric uses a decentralized mechanism based on labels. Services need to register themselves with Traefik. This is accomplished by specifying a set of labels in the service manifest to instruct Traefik which routes to forward. Further labels are available to e.g. specify rewrites of the HTTP request, or the application of additional headers. These labels are managed by the Service Fabric platform and made available to Traefik through its provider [CG18]. Program 7.21 shows how the allReady main service registers with Traefik to receive requests to the `/main` route.

As every other microservice, Traefik is deployed within a container specified by an application and a service manifest. It serves as the single entrypoint of the allReady application, forwarding all incoming traffic from the layer 4 load balancer to the cluster-internal microservices. Figure 7.3 gives an overview of the cluster architecture. On every cluster node a Traefik instance is deployed. In case an instance becomes unavailable, the Azure Load Balancer stops forwarding requests to this particular instance. To check for availabil-

[10]https://docs.microsoft.com/en-us/rest/api/servicefabric/sfclient-index

ity, the Azure Load Balancer periodically sends health checks to the node, addressing the port 8081 used by Traefik. Traefik is deliberately configured to not use port 80 for incoming HTTP requests: as explained above, most microservices in the cluster use port 80 to enable internal service discovery based on the DNS service. Traefik using the same port as other services would result in these services receiving the ALB health check. If one of these services also happens to provide an identical health check path, Azure Load Balancer will erroneously forward requests to it.

Removing a node, and with it the Traefik instance, from the rotation of the Azure Load Balancer has no effect on the availability of the back end services on that node. This is because the load balancing of Traefik instances is not restricted to destinations on the same node. For the sake of brevity, these routing possibilities are not depicted in Figure 7.3.

Within the allReady application the responsibilities of the Traefik reverse proxy are enforcing HTTPS, TLS termination, and routing requests to services depending on the path, which includes checking the health endpoint of a service to verify if it is ready to receive traffic (see Subsection 7.5.2).

A default configuration for Traefik is applied with a static TOML file that is loaded at startup. In order to query service information with the associated labels, Traefik must be pointed to the location of the Service Fabric Client REST API. For executables running natively on Service Fabric this is localhost:19080. Owing to Traefik running in a container, the Service Fabric Client REST API cannot be reached at localhost. This is solved by pointing Traefik to the public Azure Load Balancer that, in its default configuration, forwards requests on port 19080 to the Service Fabric management endpoint. Since the endpoint requires the above mentioned certificate for authorization, this approach is satisfactory from a security perspective for this prototypical implementation. Different cluster environments of course have different URLs of their management endpoints. Therefore, it is necessary to provide a parameterizable Docker image. In addition to the Traefik configuration using a TOML file, it is possible to override configuration values using identically named arguments. The Linux Dockerfile in Program 7.22 shows how this mechanism is used: at first, the container environment variable `MGMT_URL` is set, which is then applied to the Traefik configuration.

To deploy a Traefik container on Linux, an official base image is provided on Docker Hub[11]. For Windows Nano Server (nanoserver:1803), no such base image is available, thus a custom image has to be created from scratch. Traefik is based on the Go programming language; when executing Traefik

[11] https://hub.docker.com/_/traefik

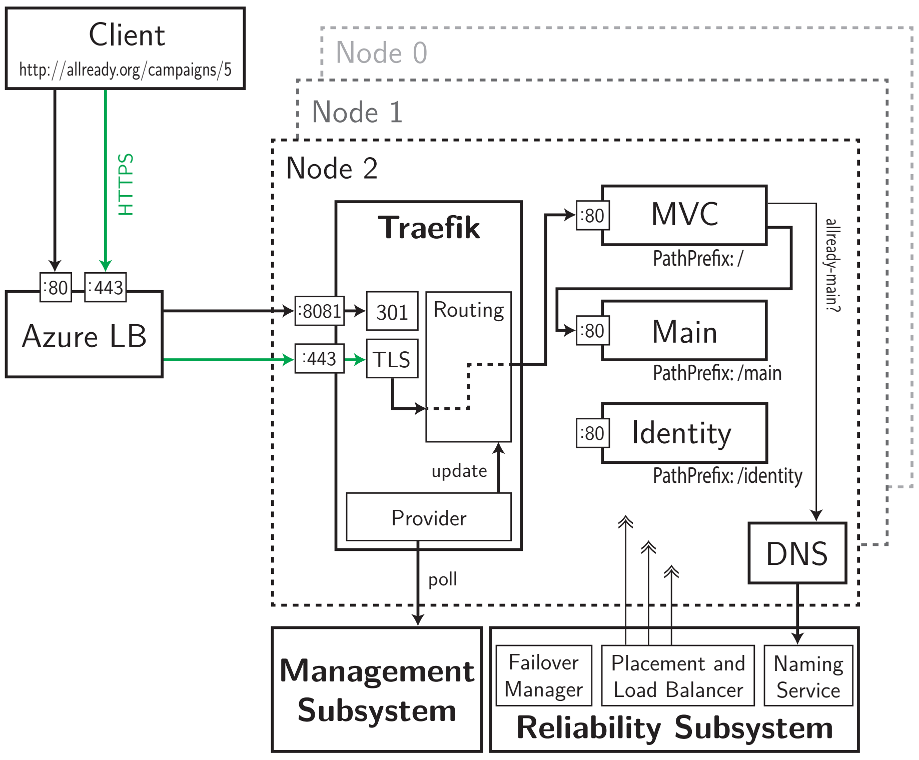


Figure 7.3: Simplified allReady architecture with Traefik as a reverse proxy; this demonstrates the process of an external client sending an HTTP request, steps I. and II. happen independently of the client request:

I. All services that provide an HTTP entrypoint (MVC, main and identity) indicate which route they will serve by specifying the `PathPrefix` label. Services are scheduled by the Placement and Load Balancer.

II. The provider component within Traefik periodically polls the management subsystem. The management subsystem returns the cluster state including the `PathPrefix` labels.

1. The client sends a HTTP request to `http://allready.org/campaigns/5`.
2. The Azure Load Balancer receives the request at port 80 and forwards it to port 8081 where Traefik is listening on.
3. To enforce HTTPS, Traefik responds with code `301 Moved Permanently`.
4. The client requests the same URL but now using HTTPS.
5. The Azure Load Balancer receives the request at port 443 and again forwards it to port 443 where Traefik is listening on.
6. Traefik decrypts the TLS connection and forwards the request to the MVC service that has registered itself for the default route.
7. The MVC service queries the DNS service for the main service location.
8. The MVC service sends an HTTP request to the main service gathering the required data to render the view.
9. The main service responds with the data and subsequently each service returns to their caller until the client receives the requested web page.

```
 1 FROM traefik:v1.7.12-alpine
 2 WORKDIR /traefik
 3
 4 COPY cert/servicefabric.crt servicefabric.crt
 5 COPY cert/servicefabric.key servicefabric.key
 6 COPY traefik.toml .
 7 EXPOSE 443
 8 EXPOSE 8080 # dashboard
 9 EXPOSE 8081
10
11 ENV MGMT_URL http://localhost:19080
12 CMD traefik --configfile=traefik.toml --servicefabric.clustermanagementurl=MGMT_URL
```

Program 7.22: Dockerfile for Linux to build the Traefik image for Service Fabric; line 12 overrides the cluster management URL depending on the environment

```
FROM mcr.microsoft.com/windows/servercore:1803 as servercore
FROM mcr.microsoft.com/powershell:nanoserver-1803
COPY --from=servercore /windows/system32/netapi32.dll /windows/system32/netapi32.dll
...
```

Program 7.23: Partial Dockerfile for Windows to import the missing DLL

in a Windows container, Go requires the system library `netapi32.dll` but this library has been removed from nanoserver:1803 in order to reduce the image size. To solve this problem, `netapi32.dll` is copied from the larger Windows Server Core image, as can be seen in Program 7.23.

Traefik is configured to force clients to use a secure connection; therefore, it has to listen on a port for HTTP (8081) and a port for HTTPS (443). When Traefik receives an unsecured HTTP request, it responds with the status code `301 Moved Permanently` and instructs the client to use the HTTPS protocol. For incoming HTTPS requests, Traefik takes over the termination of the connection and then forwards the requests to the back end services via HTTP.

On Service Fabric, Traefik itself does not support automatic issuance of certificates with Let's Encrypt; this would require all Traefik instances to be coordinated by running in cluster mode, which is in beta state for version 1.7. To enable instances forming a Traefik cluster, an additional key value store is mandatory to be deployed on Service Fabric [Con]. Storing the necessary data for the issuance process in a flat file system, like a volume backed by a central Azure Storage, is explicitly advised against due to possible conflicts [Tom18]. Since this would exceed the scope of this prototype, a self-signed certificate is used to enable encrypted communication.

7.9 Enabling continuous deployment

Continuous deployment is a fully automated process that enables shipping small iterative changes without the need for human approval before the deployment. This section discusses the characteristics of the orchestration platforms that facilitate continuous deployment. These characteristics are discussed in relation to the sequential phases of a deployment pipeline: commit, build, staging and production.

The commit phase is about facilitating a rapid development environment and a fast feedback loop for the developer. This concerns the local development environment as well as tooling that enables the developer to quickly test their code in a cloud environment before committing to source control. This thesis focuses on Windows as the developer environment. Apple's macOS is not considered.

The build phase concerns the pipeline itself and how to build the Docker images rather than the orchestration platform. Therefore, this phase is not included in the comparison.

The staging phase of a deployment pipeline covers how microservices are deployed to real environment in the cloud. Depending on the number and structure of development teams, multiple environments might be used. Running them all in a single cluster is an option to save infrastructure costs. Meticulous resource governance must ensure these environments do not interfere with each other.

In the production phase it is important to maintain high availability during the rollout of new features. To mitigate the risk of downtime, different rollout strategies exist to deploy new versions of a microservice: blue-green deployment, where both versions coexist with the same number of instances until all traffic is switched over from the old to the new version; canary deployment, where instances of the new version are gradually increased while old instances are removed simultaneously; or A/B testing, where the new version is only made available to a fixed group of end users, requiring user-based routing.

7.9.1 Azure Kubernetes Service

To group cohesive resource objects, e.g. a deployment, a service and a horizontal pod autoscaler, they can be bundled into a single package using Helm. Helm is an open source package manager supported by the Cloud Native Computing Foundation. A Helm package, also called a chart, can bundle all resources required by a microservice alongside a default configuration, and publish them in a chart repository. Various charts for ready-to-use open source components are publicly available via Helm. To operate allReady in AKS multiple components are installed using public Helm charts: the RabbitMQ message bus with built-in support for high availability, Redis in a master-replica combination, and the nginx ingress controller with the integrated cert-manager component.

Contrary to Kubernetes itself, Helm allows to automatically roll back a microservice upgrade if it fails [Hel19]. Microsoft is working on the integration of Helm charts into Azure Container Registry (ACR) to enable hosting private charts within ACR [Mic18i].

Commit phase

Depending on the developer's operating system there are different tools to run a Kubernetes cluster on their local machine. On Windows two options are available: minikube or Docker for Windows with integrated Kubernetes. Both can be deployed as single node clusters but have some differences. The use of Docker for Windows requires the underlying type 1 hypervisor Hyper-V [Docd], whereas minikube can be installed either on Hyper-V or a type 2 hypervisor like VirtualBox [Kub19r]. The disadvantage of Docker for Windows is that it comes with a pre-installed version of Kubernetes that cannot be changed [Doca].

In a microservices architecture there are many different parts potentially developed by many different teams. Although a microservice is independent of others, integrating several microservices to execute a business process is essential and might require a production-like multi-node cluster. The Microsoft product Azure Dev Spaces enables developers to deploy their code directly into an AKS cluster. This cluster runs all stable microservices of the entire application and provides an approach to quickly integrate code changes during development. Azure Dev Spaces utilizes the concept of namespaces and creates a dedicated namespace for every dev space. Each developer can have their own dev space allowing all developers to work on the same AKS cluster without interfering with each other. The basic concept of Azure Dev Spaces is the injection of sidecar containers that handle traffic routing and image building. The integrated building of Docker images facilitates rapid iterations by removing the need to build and push images into a separate container registry [Mic19d].

Staging phase

As already discussed, Kubernetes provides namespaces to separate a cluster in different environments. Not only does this help to isolate customers in a single cluster but also allows to provision several staging environments in the same cluster and individually restrict their resource consumption. This facilitates cost-efficient resource utilization for non-production environments.

Production phase

DNS names are only assigned to services but not to deployments. This decoupling allows two parallel existing deployment objects for a single microservice without causing DNS conflicts. A blue-green deployment can be executed by provisioning the new pods in the first step and then switching either by updating the service object, or the ingress resource.

For a side by side deployment of two microservice versions that both receive traffic, a single service object acts as a dispatcher and forwards requests to pods of both deployments. This mechanism can be utilized to implement a basic canary deployment strategy because traffic is automatically distributed between the pods of the deployments. The distribution ratio cannot be configured but rather is dependent on the number of pods of each deployment. For example, to let the new deployment receive 10 percent of the requests, the ration of old to new pods must be 9 to 1. To achieve an even smaller percentage, the total number of pods must be increased further, leading to a higher resource allocation. Due to the large ecosystem of Kubernetes, several third-party solutions are available to implement a more sophisticated load balancing. Depending on the specific needs, different solutions might be the best fit. Some exemplary solutions are a Traefik ingress controller[12], Flagger integrated into the standard nginx ingress controller[13], or a service mesh like Istio[14].

[12]https://docs.traefik.io/v1.7/user-guide/kubernetes/#traffic-splitting

[13]https://docs.flagger.app/usage/nginx-progressive-delivery

[14]https://istio.io/docs/concepts/traffic-management/

Without installing third-party solutions, only simple load balancing strategies can be applied. This means, it is not possible to bind a user to a particular deployment version. As a result, A/B testing also requires a third-party solution.

7.9.2 Service Fabric

Service Fabric applications need to be packaged before they are deployed to the cluster. Packages are not published to an external store but directly pushed to the Service Fabric cluster. This impedes the reusability of packages across clusters since Microsoft does not provide a dedicated package catalog for Service Fabric. However, the tool SFNuGet implements a packaging approach that allows to provide Service Fabric applications as NuGet packages [Mic18j]. As of today only 9 Service Fabric packages are publicly available on nuget.org [Micj]. Still, using existing NuGet infrastructure to provide company-internal packages can improve the development process.

Commit phase

For development purposes a Service Fabric cluster can be run on a local machine. A local Service Fabric environment can be installed on Windows as well as Linux. Natively, a local development cluster does only support containers based on the same operating system. Since this limitation would be a detrimental factor for the development of .NET Core Linux containers on Windows machines, Microsoft provides a container image that runs a Service Fabric Linux cluster on a Windows operating system [Mic17i]. This allows developers to use the Visual Studio IDE on Windows while targeting Service Fabric on Linux. In order to deploy containerized microservices to Service Fabric on Windows, Windows containers are mandatory. For local development, Windows containers must be built and run with Docker for Windows [Mic19h].

From a developer perspective, errors that occur during service deployment are not helpful. Especially with regard to containers, error messages are cryptic and don't help finding the root cause. For example, using `/bin/sh` shebang in a shell script, which is not available on Service Fabric on Linux, will result in only the following error message by the hosting subsystem: "The process/-container terminated with exit code:134." Or, if a port is already bound by a service in nat networking mode, the deployment of another service trying to bind the same port will fail. The hosting subsystem will indicate the port conflict with the vague error description `FABRIC_E_INVALID_OPERATION`.

Staging phase

Deploying multiple environments in a single cluster is only possible with several limitations. It is not easily possible to restrict an environment from accessing services residing in another environment. Furthermore, DNS names have to be coordinated and must include an environment identifier in order to differentiate between a service in different environments. Also, as discussed in Subsection 7.4.2, resource governance must be implemented on the service level; this introduces a higher susceptibility to errors.

To test the fault tolerance of a release before it is deployed to production, Service Fabric provides an API for the testability subsystem that allows to execute scenarios of failing microservices or infrastructure. Examples for these kind of events in a scenario are ungracefully restarting nodes, or terminating code packages [Mic18c] with the running container. After completing a scenario test run, a report is generated that can be included in the deployment pipeline to decide whether the release is ready for production deployment.

Production phase

Since Service Fabric itself does not provide an API gateway, it therefore does not enable more complex rollout strategies than its rolling upgrade procedure. An alternative is to utilize Traefik again to route the incoming traffic. The labels of a service to register with Traefik can be updated at runtime via the Service Fabric Client REST API. In order to execute a blue-green deployment, two versions are deployed. By increasing the value of the `traefik.frontend.priority` label on the new service version, it starts receiving all traffic. To accomplish a canary deployment strategy, Traefik can balance requests between two service versions by specifying the `traefik.backend.group.weight` label. The weight of each version defines the ratio. To gradually increase the weight of the new version, no mechanism is implemented, instead a manual update via the API is required. For A/B testing, Traefik allows to apply a regular expression on request headers. By matching on a cookie or other header values, users can be divided into different groups [CG18; Con19].

7.10 Load testing

As the final criteria, performance metrics of all three platforms are compared. For this evaluation, several load tests that simulate a constant number of parallel users are executed against each platform. These tests run for a constant time of five minutes each and are staggered in their number of users.

In order to ensure a high level of comparability between the platforms, the following measures are taken:

- All orchestration platforms operate in a cluster of three virtual machines. Each virtual machine is of type Standard_D1_v2[15], which has 1 vCPU and 3.5 GiB memory. To facilitate a replicable comparison, auto-scaling of virtual machines is not enabled.
- Although Azure Kubernetes Services and Service Fabric utilize Azure Disks differently, both use the same Standard HDD Managed Disks with up to 500 IOPS and up to 60 MB/second per disk[16].
- To mitigate the database becoming a possible bottleneck, an oversized S2[17] instance with 50 DTUs is provisioned.
- As mentioned in Subsection 7.3.3, current versions of Redis are not developed for Windows anymore. In order to keep all three platform implementation as similar as technically possible, every platform integrates Redis version 3.2, which is the last Windows release, as its storage for operational data.

Goal of the load tests is to model a realistic load. Since allReady has never been released to the public, no secondary data is available on how the application is used. To approximate real behavior, three personas are modeled, each representing a different type of user:

- **Website visitor:** A visitor that uses a browser to navigate through the allReady application. They view running campaigns related to a disaster and events corresponding to the campaigns. A website user does not log in and does not visit protected areas. The primarily targeted microservice is the MVC service, which delegates to the main service.
- **Mobile app visitor:** This visitor uses the allReady mobile app on their smartphone. They only view information but do not update it. The app

[15]https://docs.microsoft.com/azure/virtual-machines/windows/sizes-general#dv2-series
[16]https://azure.microsoft.com/pricing/details/managed-disks
[17]https://docs.microsoft.com/azure/sql-database/sql-database-service-tiers-dtu

	Website visitor	Mobile app visitor	Administrator
Ratio	4	6	1
Rel. distribution	36.36 %	54.55 %	9.09 %

Table 7.1: Workload distribution amongst all personas

sends requests directly to the `/api` path. These requests are received via the REST API of the main service, which is the primarily targeted microservice in this scenario.

- **Administrator:** The administrator is responsible for updating information about emergency aid: organizations, campaigns and events. Furthermore, their responsibility is to manage other registered users and grant them necessary permissions. The administrator logs in using the identity microservice and interacts with the website provided by the MVC service. Thus, an administrator's actions target all microservices.

Since in case of a real disaster the workloads generated by each of these three personas would not be equally distributed in size, a ration is modeled into the load tests. In order to approximate a realistic distribution, the personas are first divided into two groups: Visitors that only use the application to view information during a disaster, and administrators that coordinate measures to deal with the disaster. Due to the lack of secondary data, the total workload is split at a ration of 10 to 1 between visitor requests and administrator requests. To further distinguish between visitor personas, the visitor workload is split at a ratio of 4 to 6 between website visitors and mobile app visitors. Table 7.1 shows the resulting relative workload distribution amongst all personas.

For load generation a constant number of virtual users is created, whereas each virtual user is an object of a persona type. A virtual user executes the workflow defined by the persona. When a virtual user has finished their workflow, a new virtual user is created. Each persona has a predefined list of URLs that are called in random order. An exception are sequential requests that are required for a certain action, for example the authorization code flow of OpenID Connect a virtual user executes to log in. A virtual user always sends one request after another, i.e. during a load test, the number of parallel requests the platform has to handle does not exceed to the number of virtual users.

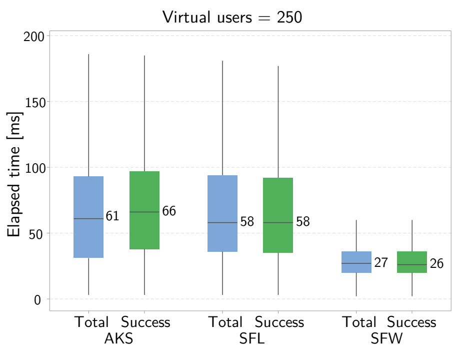
To model a realistic behavior, a virtual user waits for a certain time after receiving a response before they execute the next request. This interval is called think time. A constant think time would result in an oscillating number of parallel requests on the time axis: all virtual users would simultaneously send their first request and after receiving the response they would wait in parallel until sending the next request at the same time. Only over time the oscillation of parallel requests would level off due to different response times. To prevent this artificial densification, think time is uniformly distributed by a random number generator within an interval of 0 to 20 seconds.

Load tests are staggered in their number of parallel virtual users and are executed for a constant time of five minutes each. For test runs are executed against each platform, simulating the following numbers of parallel users: 250, 500, 1000 and 2000. These tests are expected to produce error conditions. At the start of each test, the orchestration platform is immediately impacted by the high load as all virtual users start sending requests. For an application like allReady, such impacts are common because it is designed to provide instant help and information in case of a disaster. In a scenario where a disaster strikes without prior warning, the population is alerted by civil protection alarm. As a result, people instantaneously access allReady upon hearing the sirens.

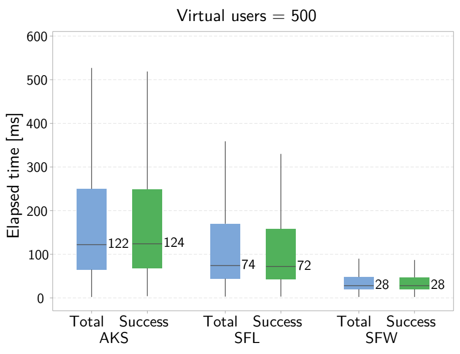
7.10.1 Response time

From a user perspective, the most important metric is the time it takes to complete a request. This metric is called response time in the sense of *time to last byte*; in other words, the time it takes from sending a request until the user receives the last byte of the response. Figure 7.4 displays the comparison of the response times as a box plot. For the response time, a distinction is made between successful responses and total responses, which also include errors. Each chart compares the platforms Azure Kubernetes Service (AKS), Service Fabric on Linux (SFL) and Service Fabric on Windows (SFW) with regard to the predefined number of virtual users.

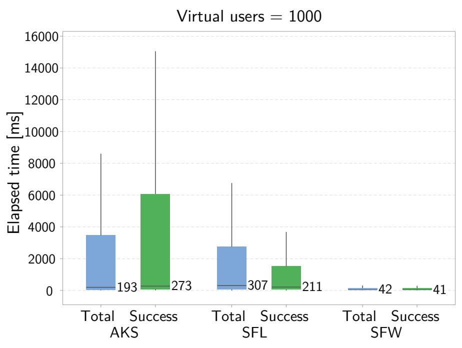
Under the load of 250 virtual users (Figure 7.4a), all three platforms respond within a reasonable time. AKS and SFL show similar results but SFW performs best. Under a higher load at 500 and 1000 virtual users (Figures 7.4b and 7.4c respectively), the response times of AKS and SFL increase significantly. Only SFW achieves a median response time under 50 ms at 1000 virtual users. Although AKS and SFL show a varying response time at 1000 virtual users, they still respond to 50 percent of the requests within a tolerable amount of time. As displayed in Figure 7.4c, only SFW is able to



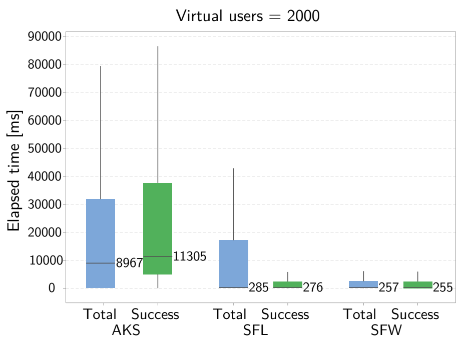
(a)



(b)



(c)



(d)

Figure 7.4: Box plots of response time distribution; the median value is displayed alongside each box; outliers are omitted to reduce noise

serve requests from 2000 virtual users in an appropriate time. Remarkable is the performance of SFL at 2000 virtual users: although a great portion of the total requests take several seconds to be returned, the successfully completed responses are processed within a tolerable time frame.

7.10.2 Throughput of successful responses

Throughput is the amount of data sent by the virtual user to the application and is measured in KiB/s. A main goal of allReady is to provide as many users with information about an emergency as possible. Therefore, for the platform comparison, only successfully completed requests are taken into account, otherwise failing responses would skew the result.

Comparing the throughput of each orchestration platform in Figure 7.5,

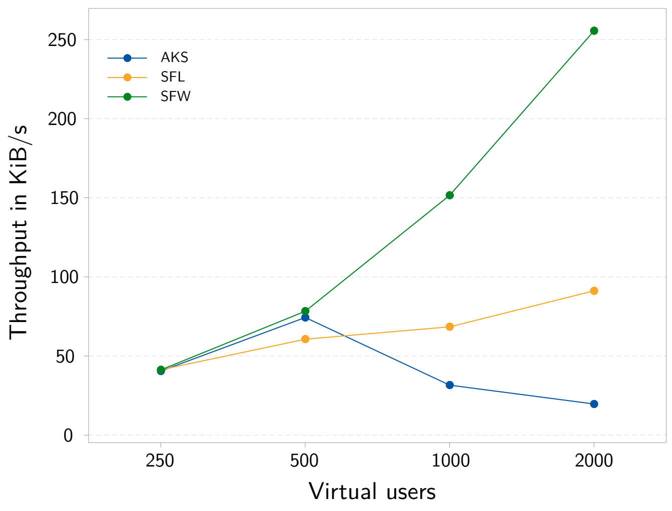


Figure 7.5: The throughput of successful responses

SFW performs best again. Doubling the number of virtual users results in almost twice the troughput. This is a logical outcome because increasing the load only has minimal effect on the response time and, as explained in the following Subsection 7.10.3, does not lead to a higher error rate.

7.10.3 Error rate

Another important factor, especially for applications such as allReady that need to be reliable in case of an emergency, is how many requests can be handled successfully. This factor is expressed by the diametrical opposite in the form of the error rate. Concerning the relative error rates in Figure 7.6, AKS and SFL struggle to respond correctly when under high load. Only SFW is able to keep the error rate under one percent.

By considering only the error rates, AKS seems to perform worse than SFL. In Figure 7.7, this error rate is set in contrast to the number of successful requests. Especially for 500 and 1000 parallel virtual users (Figures 7.7b and 7.7c respectively) AKS returns more successful responses in absolute numbers than SFL, although it has a higher error rate. For the allReady application, this means that AKS provides more users with important information in a real disaster situation. In general, it can be observed that under low load, SFL performs better than AKS (Figure 7.7a) but at the highest load, AKS achieves a better error rate and returns more successful responses (Figure 7.7d). Again, neither AKS nor SFL are able to match the

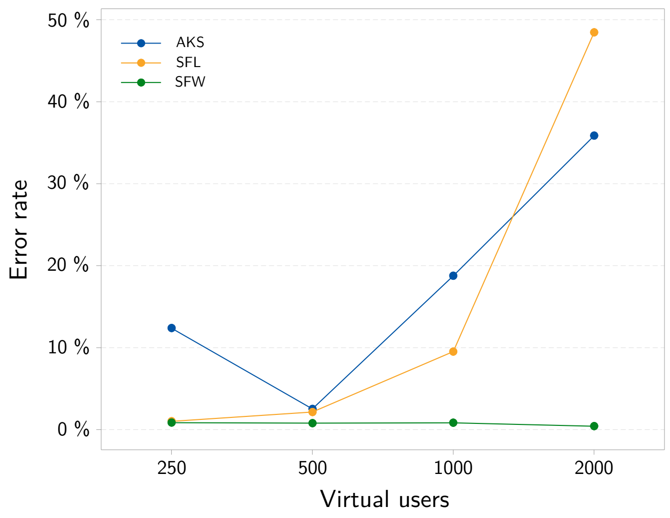


Figure 7.6: The relative error rate

performance of SFW with regard to error rates. While handling requests of 2000 virtual users, SFW operates with a minimal error rate of 0.4 %, whereas AKS and SFL have error rates of 35.9 % and 48.5 % respectively.

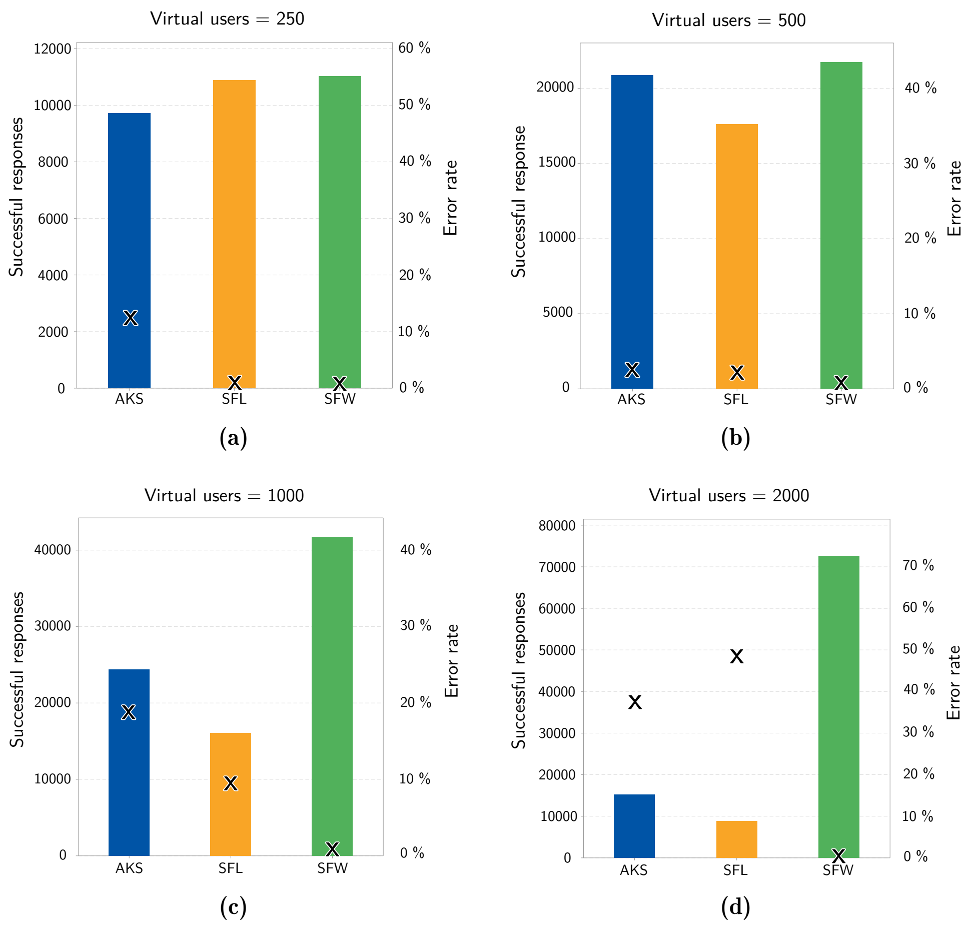


Figure 7.7: The bar charts represent the absolute numbers of successful responses; the relative error rate is indicated by the X

Chapter 8

Closing Remarks

Introducing microservices as an architecture pattern brings many benefits but also has a significant impact on the complexity of the application. This complexity is shifted from the implementation level of business logic to the higher level of application architecture. An application architecture that is built adhering to the model of eventual consistency can greatly improve the performance of distributed systems but requires every microservice to handle various faults of other systems or services.

It is a fallacy that an application can be implemented using totally decoupled and independent microservices. Most business processes involve multiple microservices that have to work together. Due to the decoupling they use asynchronous communication patterns that might lead to inconsistencies in the overall application state. This requires non-trivial coordination between microservices to compensate the inconsistencies. With regard to the transformed allReady architecture, several boundaries emerge that require the resulting microservices to coordinate. In order to handle failures during communication, many mitigation patterns are implemented. The general concept of these implemented patterns is mostly the same for different microservices but differs in their details. This leads to large portions of boilerplate code implemented in every microservice.

Despite the enormous increase in complexity, the microservices pattern enables several valuable benefits. The most important one is scalability. Depending on the workload, infrastructure can be provisioned to meet current demands. When the demand drops back to the initial state, infrastructure can be released again. In order to fully profit from this dynamic infrastructure provisioning, applications are deployed on infrastructure hosted by a cloud provider. The cloud provider offers mechanisms to dynamically create the required infrastructure resources, and only charges for allocated resources. In particular, applications with large fluctuations in workload, such as allReady, can keep the basic costs down by only provisioning minimal infrastructure required to handle average traffic. In case of rapidly increasing traffic, more infrastructure can be provisioned automatically to meet the resources requirements.

An orchestration platform represents the middle layer between application and cloud infrastructure. It monitors the cluster state and instructs the cloud provider to make changes to the underlying infrastructure. The compared container orchestration platforms show rather large difference in their range of functions. Service Fabric lacks essential functionalities, which impedes parallel development of microservices by multiple teams. Since Service Fabric supports running non-containerized executables alongside containers, it is very well suited for lift-and-shift scenarios where it is not possible to containerize every part of an application. Contrary, AKS does only allow services running inside a container. The availability of Reliable Collections within Windows containers is a unique feature that can provide an alternative to the cumbersome disk provisioning. However, this means a vendor lock-in, and therefore couples the application to the orchestration platform. Moreover, Service Fabric on Windows brings great benefits regarding performance and high availability.

Although there are fields of applications that certainly profit from the characteristics of Service Fabric, the ecosystem of Kubernetes makes it superior in most aspects. For greenfield projects or applications that can be containerized without effort, Kubernetes is the preferred choice. Moreover, a microservices architecture must also be implemented within the organization of a company. Kubernetes provides the tools to separate independent development teams and minimize coordination overhead.

References

[AB] Brock Allen and Dominick Baier. *Deployment.* URL: http://docs.identityserver.io/en/latest/topics/deployment.html#operational-data (visited on 2019-11-18) (cit. on p. 52).

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". *IEEE Software* 33.3 (2016), pp. 42–52 (cit. on p. 28).

[Bre00] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing.* PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 7–. URL: http://doi.acm.org/10.1145/343477.343502 (cit. on p. 24).

[Bro19] Taylor Brown. *Announcing the preview of Windows Server containers support in Azure Kubernetes Service.* May 17, 2019. URL: https://azure.microsoft.com/en-us/blog/announcing-the-preview-of-windows-server-containers-support-in-azure-kubernetes-service (visited on 2019-11-18) (cit. on p. 50).

[Bur19] Brendan Burns. *Azure Kubernetes Service (AKS) GA – New regions, more features, increased productivity.* June 13, 2019. URL: https://azure.microsoft.com/en-us/blog/azure-kubernetes-service-aks-ga-new-regions-new-features-new-productivity (visited on 2019-11-22) (cit. on p. 56).

[CG18] Joni Collinge and Lawrence Gripper. *Intelligent routing on Service Fabric with Træfik.* Microsoft. April 5, 2018. URL: https://techcommunity.microsoft.com/t5/Azure-Service-Fabric/Intelligent-routing-on-Service-Fabric-with-Tr-230-fik/ba-p/791290 (visited on 2019-12-10) (cit. on pp. 91, 99).

[Cit+15] Jürgen Cito et al. "The making of cloud applications: an empirical study on software development for the cloud". *Proceedings* *of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015* (2015), pp. 393–403. URL: http://dl.acm.org/citation.cfm?doid=2786805.2786826 (cit. on p. 1).

[Cle14] Toby Clemson. *Testing Strategies in a Microservice Architecture.* November 18, 2014. URL: https://martinfowler.com/articles/microservice-testing (visited on 2018-05-01) (cit. on p. 30).

[Con] Containous. *Clustering / High Availability (beta).* URL: https://docs.traefik.io/v1.7/user-guide/cluster (visited on 2019-12-10) (cit. on p. 94).

[Con19] Containous. *Basics - Traefik.* March 15, 2019. URL: https://docs.traefik.io/v1.7/basics (visited on 2019-12-04) (cit. on pp. 79, 99).

[Con68] Melvin E. Conway. "How do committees invent". *Datamation* 14.4 (1968), pp. 28–31. URL: http://www.melconway.com/Home/pdf/committees.pdf (cit. on p. 27).

[Din18] Sandeep Dinesh. *Kubernetes best practices: Setting up health checks with readiness and liveness probes.* Google. May 4, 2018. URL: https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-setting-up-health-checks-with-readiness-and-liveness-probes (visited on 2019-12-03) (cit. on p. 77).

[Doca] Docker. *Deploy on Kubernetes.* URL: https://docs.docker.com/docker-for-windows/kubernetes (visited on 2019-12-13) (cit. on p. 96).

[Docb] Docker. *Docker runtime execution options.* URL: https://docs.docker.com/engine/reference/commandline/dockerd/#docker-runtime-execution-options (visited on 2019-12-06) (cit. on p. 83).

[Docc] Docker. *Dockerfile reference.* URL: https://docs.docker.com/engine/reference/builder (visited on 2019-12-05) (cit. on p. 80).

[Docd] Docker. *Install Docker Desktop on Windows.* URL: https://docs.docker.com/docker-for-windows/install (visited on 2019-12-13) (cit. on p. 96).

[Doc18] Docker. *Docker for Azure persistent data volumes.* March 13, 2018. URL: https://docs.docker.com/docker-for-azure/persistent-data-volumes (visited on 2019-12-01) (cit. on p. 72).

[Dra+17] Nicola Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering.* Ed. by Manuel Mazzara and Bertrand Meyer. Springer International Publishing, 2017, pp. 195–216. URL: https://doi.org/10.1007/978-3-319-67425-4_12 (cit. on pp. 10, 11).

[Etc19] etcd Development and Communities. *etcd version 3.4.0*. August 30, 2019. URL: https://etcd.io/docs (visited on 2019-11-21) (cit. on p. 55).

[Eva03] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, 2003 (cit. on pp. 14, 15).

[Fowa] Martin Fowler. *Microservices and the First Law of Distributed Objects*. URL: https://martinfowler.com/articles/distributed-objects-microservices.html (visited on 2018-04-13) (cit. on p. 13).

[Fowb] Martin Fowler. *Microservices Resource Guide*. URL: https://martinfowler.com/microservices/ (visited on 2018-01-07) (cit. on p. 12).

[Fow11] Martin Fowler. *TolerantReader*. May 9, 2011. URL: https://martinfowler.com/bliki/TolerantReader.html (visited on 2018-04-17) (cit. on p. 23).

[Git] GitHub, Inc. *Commits · microsoft/service-fabric*. URL: https://github.com/microsoft/service-fabric/commits/master (visited on 2019-11-24) (cit. on p. 58).

[GT17] J. P. Gouigoux and D. Tamzalit. "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. April 2017, pp. 62–65 (cit. on pp. 11, 17).

[Hel19] Helm Authors. *Helm upgrade - Options*. May 16, 2019. URL: https://v2.helm.sh/docs/helm/#options-43 (visited on 2019-12-13) (cit. on p. 96).

[Hor17] Christian Horsdal. *Microservices in . NET Core*. Manning, 2017 (cit. on pp. 18, 19).

[Jan19] Deepak Jangid. *ServiceFabricVolumeDisk on private cluster · Issue #1380 · Azure/service-fabric-issues*. October 29, 2019. URL: https://github.com/Azure/service-fabric-issues/issues/1380#issuecomment-547503036 (visited on 2019-12-05) (cit. on p. 72).

[JBS15] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. URL: https://tools.ietf.org/html/rfc7519 (cit. on p. 37).

[Kak+18] Gopal Kakivaya et al. "Service Fabric: A Distributed Platform for Building Microservices in the Cloud". *Proceedings of the 13th EuroSys Conference, EuroSys 2018* (April 2018), 33:1–33:15 (cit. on p. 59).

[Kha17] A Khan. “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application”. *IEEE Cloud Computing* 4.5 (September 2017), pp. 42–48 (cit. on pp. 47, 53).

[Kil16] Tom Killalea. “The hidden dividends of microservices”. *Communications of the ACM* 59.8 (2016), pp. 42–45. URL: http://dl.acm.org/citation.cfm?doid=2975594.2948985 (cit. on p. 28).

[Kub19a] Kubernetes Authors. *Assigning Pods to Nodes*. November 1, 2019. URL: https://kubernetes.io/docs/concepts/configuration/assign-pod-node (visited on 2019-11-30) (cit. on p. 70).

[Kub19b] Kubernetes Authors. *Building large clusters*. June 12, 2019. URL: https://kubernetes.io/docs/setup/best-practices/cluster-large (visited on 2019-11-29) (cit. on p. 67).

[Kub19c] Kubernetes Authors. *Cluster Networking*. December 6, 2019. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking (visited on 2019-12-07) (cit. on p. 84).

[Kub19d] Kubernetes Authors. *Concepts Underlying the Cloud Controller Manager*. November 14, 2019. URL: https://kubernetes.io/docs/concepts/architecture/cloud-controller (visited on 2019-11-23) (cit. on p. 54).

[Kub19e] Kubernetes Authors. *Configure a Security Context for a Pod or Container*. September 19, 2019. URL: https://kubernetes.io/docs/tasks/configure-pod-container/security-context (visited on 2019-12-05) (cit. on p. 81).

[Kub19f] Kubernetes Authors. *Deployments*. November 22, 2019. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment (visited on 2019-11-23) (cit. on pp. 57, 77).

[Kub19g] Kubernetes Authors. *Horizontal Pod Autoscaler*. November 26, 2019. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale (visited on 2019-11-30) (cit. on p. 68).

[Kub19h] Kubernetes Authors. *Ingress*. October 18, 2019. URL: https://kubernetes.io/docs/concepts/services-networking/ingress (visited on 2019-12-08) (cit. on p. 86).

[Kub19i] Kubernetes Authors. *Ingress Controllers*. October 24, 2019. URL: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers (visited on 2019-12-08) (cit. on p. 87).

[Kub19j] Kubernetes Authors. *Kubernetes Components*. November 13, 2019. URL: https://kubernetes.io/docs/concepts/overview/components (visited on 2019-11-23) (cit. on p. 54).

[Kub19k] Kubernetes Authors. *Managing Compute Resources for Containers.* November 18, 2019. URL: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container (visited on 2019-11-30) (cit. on p. 67).

[Kub19l] Kubernetes Authors. *Persistent Volumes.* November 29, 2019. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes (visited on 2019-11-30) (cit. on p. 69).

[Kub19m] Kubernetes Authors. *Pod Lifecycle.* November 5, 2019. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle (visited on 2019-11-30) (cit. on p. 76).

[Kub19n] Kubernetes Authors. *Pod Overview.* October 23, 2019. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview (visited on 2019-11-23) (cit. on p. 56).

[Kub19o] Kubernetes Authors. *Pod Security Policies.* September 13, 2019. URL: https://kubernetes.io/docs/concepts/policy/pod-security-policy (visited on 2019-12-05) (cit. on p. 81).

[Kub19p] Kubernetes Authors. *Secrets.* November 12, 2019. URL: https://kubernetes.io/docs/concepts/configuration/secret (visited on 2019-12-05) (cit. on p. 81).

[Kub19q] Kubernetes Authors. *Service.* November 12, 2019. URL: https://kubernetes.io/docs/concepts/services-networking/service (visited on 2019-11-23) (cit. on pp. 55, 57).

[Kub19r] Kubernetes Authors. *Windows - Prerequisites.* September 19, 2019. URL: https://minikube.sigs.k8s.io/docs/start/windows/#prerequisites (visited on 2019-12-13) (cit. on p. 96).

[Lan16] Rich Lander. *Announcing .NET Core 1.0.* June 27, 2016. URL: https://blogs.msdn.microsoft.com/dotnet/2016/06/27/announcing-net-core-1-0 (visited on 2018-03-15) (cit. on p. 5).

[LF14] James Lewis and Martin Fowler. *Microservices - a definition of this new architectural term.* 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 2018-01-07) (cit. on pp. 9, 13, 15, 17).

[Mar10] Robert C. Martin. "The Single Responsibility Principle". In: *97 Things Every Programmer Should Know.* 1st ed. O'Reilly, 2010, pp. 152–153 (cit. on p. 14).

[McA+] Jimmy McArthur et al. *2018 OpenStack User Survey Report.* URL: https://www.openstack.org/user-survey/2018-user-survey-report (visited on 2019-11-21) (cit. on p. 54).

[McK19] Micah McKittrick. *Reliable Collections within Docker container · Issue #41667 · MicrosoftDocs/azure-docs*. November 18, 2019. URL: https://github.com/MicrosoftDocs/azure-docs/issues/41667#issuecomment-555107360 (visited on 2019-11-28) (cit. on p. 64).

[Mica] Microsoft. *API Management pricing*. URL: https://azure.microsoft.com/en-us/pricing/details/api-management (visited on 2019-12-09) (cit. on p. 90).

[Micb] Microsoft. *Choose between ASP.NET and ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/choose-aspnet-framework (visited on 2018-03-16) (cit. on p. 7).

[Micc] Microsoft. *Container services*. URL: https://azure.microsoft.com/en-us/product-categories/containers (visited on 2019-11-15) (cit. on p. 49).

[Micd] Microsoft. *Get started with the .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/framework/get-started/index (visited on 2018-03-15) (cit. on p. 5).

[Mice] Microsoft. *Kestrel web server implementation in ASP.NET Core*. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?tabs=aspnetcore2x (visited on 2018-03-16) (cit. on p. 7).

[Micf] Microsoft. *.NET Core application deployment*. URL: https://docs.microsoft.com/en-us/dotnet/core/deploying/index (visited on 2018-03-15) (cit. on p. 7).

[Micg] Microsoft. *.NET Core Guide*. URL: https://docs.microsoft.com/en-us/dotnet/core (visited on 2018-03-15) (cit. on p. 6).

[Mich] Microsoft. *.NET Framework system requirements*. URL: https://docs.microsoft.com/en-us/dotnet/framework/get-started/system-requirements (visited on 2018-03-15) (cit. on p. 6).

[Mici] Microsoft. *.NET Standard FAQ*. URL: https://github.com/dotnet/standard/blob/master/docs/faq.md (visited on 2018-03-16) (cit. on p. 8).

[Micj] Microsoft. *NuGet Gallery | Packages matching Tags:"SFNuGet"*. URL: https://www.nuget.org/packages?q=Tags%3A%22SFNuGet%22 (visited on 2019-12-14) (cit. on p. 98).

[Mick] Microsoft. *Overview of the .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview (visited on 2018-03-15) (cit. on p. 5).

[Micl] Microsoft. *ResourceGovernancePolicyType Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.fabric.management.servicemodel.resourcegovernancepolicytype (visited on 2019-12-01) (cit. on p. 71).

[Mic16] Microsoft. *Release 3.2.100 · microsoftarchive/redis*. July 1, 2016. URL: https://github.com/microsoftarchive/redis/releases/tag/win-3.2.100 (visited on 2019-11-27) (cit. on p. 64).

[Mic17a] Microsoft. *Bulkhead pattern*. June 23, 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead (visited on 2018-04-16) (cit. on p. 21).

[Mic17b] Microsoft. *Compensating Transaction pattern*. June 23, 2017. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction (visited on 2018-04-18) (cit. on p. 25).

[Mic17c] Microsoft. *Connect and communicate with services in Service Fabric*. November 1, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-connect-and-communicate-with-services (visited on 2019-12-09) (cit. on p. 89).

[Mic17d] Microsoft. *Reverse proxy in Azure Service Fabric*. November 3, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reverseproxy (visited on 2019-12-09) (cit. on p. 89).

[Mic17e] Microsoft. *Service Fabric architecture*. October 12, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-architecture (visited on 2019-11-24) (cit. on p. 59).

[Mic17f] Microsoft. *Service Fabric programming model overview*. November 2, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-choose-framework (visited on 2019-11-24) (cit. on p. 58).

[Mic17g] Microsoft. *Service Fabric with Azure API Management overview*. June 22, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-api-management-overview (visited on 2019-12-09) (cit. on p. 90).

[Mic17h] Microsoft. *Service movement cost*. August 18, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-movement-cost (visited on 2019-12-02) (cit. on p. 75).

[Mic17i] Microsoft. *Set up a Linux Service Fabric cluster on your Windows developer machine*. November 20, 2017. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-local-linux-cluster-windows (visited on 2019-12-14) (cit. on p. 98).

[Mic18a] Microsoft. *Containerize your Service Fabric Reliable Services and Reliable Actors on Windows*. May 23, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-services-inside-containers (visited on 2019-11-28) (cit. on p. 64).

[Mic18b] Microsoft. *Differences between Service Fabric on Linux and Windows*. February 23, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-linux-windows-differences (visited on 2019-11-27) (cit. on p. 64).

[Mic18c] Microsoft. *Induce controlled Chaos in Service Fabric clusters*. February 5, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-controlled-chaos (visited on 2019-12-14) (cit. on p. 99).

[Mic18d] Microsoft. *Introduction to Auto Scaling*. April 17, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-autoscaling (visited on 2019-12-01) (cit. on p. 71).

[Mic18e] Microsoft. *Introduction to Service Fabric health monitoring*. February 28, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-health-introduction (visited on 2019-12-03) (cit. on p. 78).

[Mic18f] Microsoft. *Run a service startup script as a local user or system account*. March 21, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-run-script-at-service-startup (visited on 2019-12-06) (cit. on p. 82).

[Mic18g] Microsoft. *Service Fabric application upgrade*. February 23, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-application-upgrade (visited on 2019-12-04) (cit. on p. 79).

[Mic18h] Microsoft. *Service Fabric container networking modes*. February 23, 2018. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-networking-modes (visited on 2019-12-07) (cit. on p. 84).

[Mic18i] Microsoft. *Use Azure Container Registry as a Helm repository for your application charts*. September 24, 2018. URL: https://docs.microsoft.com/en-us/azure/container-registry/container-registry-helm-repos (visited on 2019-12-13) (cit. on p. 96).

[Mic18j] Microsoft. *What's SFNuGet?* August 31, 2018. URL: https://github.com/Azure/SFNuGet (visited on 2019-12-14) (cit. on p. 98).

[Mic19a] Microsoft. *Azure keyvault integration with Kubernetes via a Flex Volume*. November 7, 2019. URL: https://github.com/Azure/kubernetes-keyvault-flexvol (visited on 2019-12-07) (cit. on p. 82).

[Mic19b] Microsoft. *Azure subscription and service limits, quotas, and constraints.* November 18, 2019. URL: https://docs.microsoft.com/en-us/azure/azure-subscription-service-limits (visited on 2019-12-07) (cit. on p. 67).

[Mic19c] Microsoft. *Frequently asked questions about Azure Container Instances.* April 25, 2019. URL: https://docs.microsoft.com/en-us/azure/container-instances/container-instances-faq (visited on 2019-11-15) (cit. on p. 49).

[Mic19d] Microsoft. *How Azure Dev Spaces works and is configured.* March 4, 2019. URL: https://docs.microsoft.com/en-us/azure/dev-spaces/how-dev-spaces-works (visited on 2019-12-13) (cit. on p. 97).

[Mic19e] Microsoft. *KeyVaultReference support for Service Fabric applications (preview).* September 20, 2019. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-keyvault-references (visited on 2019-12-05) (cit. on p. 83).

[Mic19f] Microsoft. *Kubernetes core concepts for Azure Kubernetes Service (AKS).* June 3, 2019. URL: https://docs.microsoft.com/en-us/azure/aks/concepts-clusters-workloads (visited on 2019-11-23) (cit. on p. 56).

[Mic19g] Microsoft. *Microservices architecture on Azure Service Fabric.* June 13, 2019. URL: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/microservices/service-fabric#choose-an-application-to-service-packaging-model (visited on 2019-12-18) (cit. on p. 62).

[Mic19h] Microsoft. *Prepare your development environment on Windows.* November 18, 2019. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-get-started (visited on 2019-12-14) (cit. on p. 98).

[Mic19i] Microsoft. *Preview - Secure your cluster using pod security policies in Azure Kubernetes Service (AKS).* April 17, 2019. URL: https://docs.microsoft.com/en-us/azure/aks/use-pod-security-policies (visited on 2019-12-05) (cit. on p. 81).

[Mic19j] Microsoft. *Scaling in Service Fabric.* August 26, 2019. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-scalability (visited on 2019-12-01) (cit. on p. 71).

[Mic19k] Microsoft. *Secrets.* July 25, 2019. URL: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-application-secret-store (visited on 2019-12-05) (cit. on p. 83).

[Mic19l] Microsoft. *Tutorial: Create a multi-container (preview) app in Web App for Containers.* April 29, 2019. URL: https://docs.microsoft.com/en-us/azure/app-service/containers/tutorial-multi-container-app (visited on 2019-11-15) (cit. on p. 49).

[Mic19m] Microsoft. *What is Azure Application Gateway?* November 23, 2019. URL: https://docs.microsoft.com/en-us/azure/application-gateway/overview (visited on 2019-12-09) (cit. on p. 90).

[New15] Sam Newman. *Building Microservices.* O'Reilly, 2015 (cit. on pp. 12–14, 16, 17, 21, 22, 25–27, 29, 42).

[Nyg07] Michael Nygard. *Release It!: Design and Deploy Production-Ready Software.* Pragmatic Bookshelf, 2007 (cit. on pp. 20, 21).

[Pos80] Jon Postel. *Transmission Control Protocol.* RFC 761. January 1980. URL: https://tools.ietf.org/html/rfc761 (cit. on p. 23).

[Pre] Tom Preston-Werner. *Semantic Versioning 2.0.0.* URL: https://semver.org/ (visited on 2018-04-17) (cit. on p. 22).

[Ram17] Subramanian Ramaswamy. *Orchestrating one million containers with Azure Service Fabric.* Youtube. September 27, 2017. URL: https://www.youtube.com/watch?v=OjhOZkql4uE (visited on 2019-11-30) (cit. on p. 70).

[Rica] Chris Richardson. *Pattern: API Gateway / Backends for Frontends.* URL: https://microservices.io/patterns/apigateway.html (visited on 2019-12-18) (cit. on pp. 23, 24).

[Ricb] Chris Richardson. *Pattern: Monolithic Architecture.* URL: http://microservices.io/patterns/monolithic.html (visited on 2018-03-17) (cit. on pp. 10, 11).

[Ric18] Chris Richardson. *Microservices Patterns: With examples in Java.* MEAP Edition. unpublished draft. Manning, 2018. URL: https://www.manning.com/books/microservices-patterns (cit. on p. 11).

[Rob06] Ian Robinson. *Consumer-Driven Contracts: A Service Evolution Pattern.* June 12, 2006. URL: https://martinfowler.com/articles/consumerDrivenContracts.html (visited on 2018-04-17) (cit. on p. 23).

[SBJ14] N. Sakimura, J. Bradley, and M. Jones. *OpenID Connect Core 1.0 incorporating errata set 1.* Tech. rep. November 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html (cit. on p. 36).

[Sen19] Athinanthny Senthil. *Service Fabric 7.0 Release*. Microsoft. November 18, 2019. URL: https://techcommunity.microsoft.com/t5/Azure-Service-Fabric/Service-Fabric-7-0-Release/ba-p/1015482 (visited on 2019-12-05) (cit. on p. 83).

[Ser18] Service Fabric Team. *Service Fabric is going open source*. March 14, 2018. URL: https://techcommunity.microsoft.com/t5/Azure-Service-Fabric/Service-Fabric-is-going-open-source/ba-p/791251 (visited on 2019-11-24) (cit. on p. 58).

[Sti15] Matt Stine. *Migrating to Cloud-Native Application Architectures*. 1997. O'Reilly, 2015. URL: http://www.oreilly.com/programming/free/migrating-cloud-native-application-architectures.csp?download=true (cit. on p. 24).

[Sys19] Sysdig, Inc. *2019 Container Usage Report*. 2019. URL: https://sysdig.com/resources/papers/2019-container-usage-report/ (cit. on pp. 46, 54).

[Tho] ThoughtWorks. *Inverse Conway Maneuver*. URL: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver (visited on 2018-05-09) (cit. on p. 28).

[Tom18] Daniel Tomcej. *Allow Traefik to function in HA with Flatfile · Issue #3594 · containous/traefik*. July 11, 2018. URL: https://github.com/containous/traefik/issues/3594#issuecomment-404217737 (visited on 2019-12-10) (cit. on p. 95).

[TWR17] Cesar de la Torre, Bill Wagner, and Mike Rousos. *.NET Microservices: Architecture for Containerized .NET Applications*. 2nd ed. Microsoft Developer Division, .NET and Visual Studio product teams, 2017. URL: https://aka.ms/microservicesebook (cit. on pp. 17, 18).

[Vau15] Steven J. Vaughan-Nichols. *Google releases Kubernetes 1.0: Container management will never be the same*. July 21, 2015. URL: https://www.zdnet.com/article/google-releases-kubernetes-1-0 (visited on 2019-11-21) (cit. on p. 54).

[Vog09] Werner Vogels. "Eventually Consistent". *Commun. ACM* 52.1 (January 2009), pp. 40–44. URL: http://doi.acm.org/10.1145/1435417.1435432 (cit. on p. 24).

[Vog16] Werner Vogels. *10 Lessons from 10 Years of Amazon Web Services*. March 11, 2016. URL: https://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html (visited on 2018-04-16) (cit. on p. 19).

[War82] Walter P Warner. *What is a Software Engineering Environment (SEE). (Desired Characteristics).* Tech. rep. ADA124280. Defense Technical Information Center, December 1982, p. 20. URL: http://www.dtic.mil/docs/citations/ADA124280 (cit. on p. 11).

[WC03] Laurie Williams and Alistair Cockburn. “Agile Software Development: It’s about Feedback and Change”. *IEEE Computer Society* (2003), pp. 39–43 (cit. on p. 27).

[Wol16] Eberhard Wolff. *Microservices: Flexible Software Architectures.* 2016 (cit. on pp. 11, 13–15, 21, 29, 30).