Communication and Deployment Microservices Architecture Patterns

Since its creation in 2015 Microservice Architecture is rising in popularity and is now a widely adopted standard. Most Microservice deployments consists of a huge number of services. Deploying them all efficiently and letting them communicate reliably is a complex task. Patterns help developers solve this common problems. This work identify common deployment and communication patterns, categorises them and presents their advantages and their disadvantages.

1 Introduction

This Introduction first defines the fundamental Concepts used in this Thesis. In Chapter two we categorize the reported patterns into categories. We define and discuss all categorized patterns in the third and fourth chapter. The discussion includes positive and negative effects on selected quality attributes. Most discussed papers are partly defined through graphic illustrations created by us.

1.1 Microservices

"*A **monolith** is a software application whose modules cannot be executed independently.*" [10]

The classical way to develop an application is to code everything in one big program. The result is a monolithic application, where all parts are dependent on each other. With bigger and more complex projects that have to serve more customers, this approach is more and more challenging. One solution to this problem which emerged in 2011[10] is the use of microservice architecture. Through this seminar, we assume the definitions for a microservice provided by Dragoni et al.

"*A **microservice** is a cohesive, independent process interacting via messages.*"

"*A **microservice architecture** is a distributed application where all its modules are microservices.*" [10]

1.2 Architecture Patterns

While developing applications, developers often face similar problems. The solutions to these problems can be abstracted and reused. We use the definitions provided by Richards et al.[25, p.6] to differentiate between architecture styles, architecture patterns, and design patterns.

"*An **architecture style** [...] describe the macro structure of a system*"

"***Architecture patterns** [...] describe reusable structural building block patterns that can be used within each of the architecture styles to solve a particular problem.*"

"*Architecture patterns, in turn, differ from **design patterns** [...] in that an architecture pattern impacts the structural aspect of a system, whereas a design pattern impacts how the source code is designed.*" [25, p.6]

2 Classification

In this paper, we focus on architecture patterns inside the architecture style microservices. Only patterns related to the types of systems on which microservices can be deployed, and patterns related to communication styles are examined. From the literature, we identified different patterns that are used in microservice architectures. These patterns can be classified into different categories. In this section, we will present the classification of the patterns.

2.1 Communication and Messaging

All Communication can be divided into synchronous and asynchronous communication. Synchronous communication involves one service sending a request and blocking up until an answer is received. The answering service halts what it is currently doing and answers immediately. In an Asynchronous communication the sending service does not wait for a response and only processed it once it arrives. The answering service finishes its current task and answers when it has time.

2.1.1 Synchronous Communication

- **Communication using Binary Protocols** 3.1 Described in:[16, 36, 1, 31, 37, 22]. Microservices call functions on remote servers the same way they would call a local function.

An Example implementation can be found at `github.com/ianrtracey/go-thrift`.

- **Point-to-Point Communication** 3.2 Described in:[16, 22, 1, 3, 37, 7]. The Pattern describes a direct, synchronous communication between two services via an API.
  An Example implementation can be found at `github.com/ChintanGohel1/jms-point-to-point-model`.

2.1.2 Asynchronous Communication

- **(Priority) Queue** 3.3 Described in:[34, 24, 28, 13, 2]. Messages get stored in a queue from where they can be consumed later.
  An Example implementation can be found at `github.com/mspnp/cloud-design-patterns/tree/main/priority-queue`.
- **API Gateway Pattern** 3.4 Described in:[24, 26, 16, 32, 22, 36, 37, 21, 7]. The requesting service calls a gateway service that forwards the request to the right recipient.

2.1.3 Communication through a Gateway

All following patterns can be categorized as sub-patterns of the API Gateway Pattern.

- **Anti-Corruption Layer (Adapter)** 3.5 Described in:[28, 27, 34, 4, 5]. API Gateway to separate Systems with different API Syntax.
  An Example implementation can be found at `github.com/aws-samples/anti-corruption-layer-pattern`.
- **Aggregator Pattern** 3.6 Described in:[24, 28, 36]. Gateway Service gets information from multiple other services.
- **Publish/Subscribe Communication** 3.7 Described in:[16, 33, 28, 37]. Messages are published to topics which can be subscribed to.
- **Ambassador** 3.8 described in:[28, 5]. Communication Proxy to handle all communication details for the service.
  An Example implementation can be found at `github.com/bbachi/k8s-ambassador-container-pattern`.
- **Back End for Frontends** 3.9 Described in:[22, 28, 36, 37]. Every Client Type has its own API Gateway.
- **Gateway Routing** 3.10 Described in:[28, 26, 34]. Hide multiple services behind one Gateway.

2.1.4 Mixed Sync/Async Communication

- **Chained/Branching Microservice Pattern** 3.11 Described in:[24]. Request travels through services.

2.1.5 Communication Styles

- **Circuit Breaker Pattern** 3.12 Described in:[24, 28, 17, 18, 7]. Stop trying to communicate after some attempts.
  An Example implementation can be found at `github.com/rubyist/circuitbreaker`.
- **Time-Outs** 3.13 Described in:[22, 8, 27]. Stop waiting for a response after a predefined time.
- **Retries** 3.14 Described in:[22, 28, 18]. If an Operation fails, it is repeated after some time.

2.2 Deployment

Six Patterns related to where microservices can be deployed were found.

- **Serverless** 4.1 Described in:[16, 26, 11, 22, 14, 7]. Small stateless functions run invoked by events.
- **Service Deployment Platform** 4.2 Described in:[26, 11, 22, 7]. Managed Infrastructure where containers or vms can be deployed.
- **Service Instance per Container** 4.3 Described in:[16, 26, 17, 6, 22, 30, 21]. Each Microservice is packed into a Container. Containers provide VM like isolation with less overhead.
  An Example implementation can be found at `github.com/microservices-patterns/ftgo-application`.
- **Service Instance per VM** 4.4 Described in:[16, 26, 23, 15, 26]. Each Microservice is packed in on Virtual Machine (VM).
- **Service Instance per Host** 4.5 Described in:[32, 26, 22]. Each Microservice runs on its own bare metal host.
- **Multiple Service Instances per Host** 4.6 Described in:[26, 32, 22]. Multiple Microservices run on one host without separation.

2.3 Quality Attributes

To discuss the patterns, eight quality attributes are used. Vale et al.[35] proposes Scalability, Performance, Availability, Monitorability, Security, Testability, and Maintainability as attributes for microservices patterns. Additionally, Russinovich et al.[29] includes some of the mentioned attributes but adds Cost optimization in their quality attributes.
We color the affected attributes red if the pattern has a negative impact on the attribute, green if the pattern has a positive impact, and gray if no impact is observed. A yellow color means that no immediate negative effect is observed, but depending on the specific implementation, it could have a negative effect.

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 1: Quality Attributes

3 Communication/Messaging Architecture Patterns Discussion

3.1 Communication with Binary Protocols (RPC) Pattern

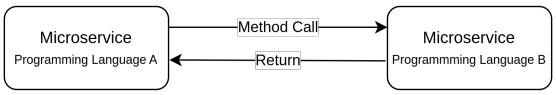

Figure 1: Communication with binary protocols Pattern

Remote Procedure Call (RPC) is a communication style where programs call functions on remote servers the same way they call local functions. Traditionally, it is only possible inside the same programming language. The idea of microservices is using the right language for each service and not like in monolith projects write everything in one language. Projects like Apache Thrift[1], SOAP[2], gRPC[3] or Google Protobuf[4] allow RPC across a wide range of languages.

3.1.1 Advantages

RPC is highly scalable[31], a feature that is particularly beneficial for microservices architecture. Furthermore, it is recognized for its efficiency[37] and good performance[1, 36]. One of the key advantages of RPC is its simplicity of implementation. Developers can use normal method calls, making the process straightforward and less complex[22].

3.1.2 Disadvantages

One drawback is the marginal performance cost associated with its use[31]. Developers may overlook the fact that network calls are not as reliable and fast as local calls, which can lead to potential issues[22]. The process of updating call specifications can be challenging, adding to the complexity of using RPC[22]. The inability to proxy responses is another limitation of this communication style[1].

[1] https://thrift.apache.org/
[2] https://docs.oracle.com/cd/A97335_02/integrate.102/a90297/overview.htm
[3] https://grpc.io/
[4] https://github.com/protocolbuffers/protobuf?tab=readme-ov-file

3.1.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 2: Binary protocols Quality Attributes

3.2 Point-to-Point (Synchronous API) Communication Pattern

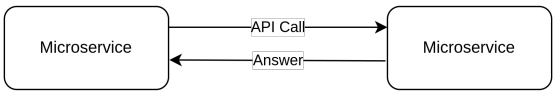

Figure 2: Point-to-Point Pattern

The Pattern describes a direct, synchronous communication between two services via an Application Programming Interface(API).
REST over HTTP API Specifications, GraphQL[5] or OpenAPI[6] can be used to implement it.

3.2.1 Advantages

This pattern is straightforward to plan and implement[3] and offers good Maintainability[1]. Using a REST Style API allows simple and large Scale caching[22][37, p.176].

3.2.2 Disadvantages

With many Services, the complexity of the communication increases, and with it the performance degrades[3]. This pattern offers lower Performance and latency than using binary protocols[22].

3.2.3 Quality Attributes

[5] https://graphql.org/
[6] https://spec.openapis.org/oas/latest.html

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 3: Point-to-Point Quality Attributes

3.3 Asynchronous (Priority) Queue Pattern

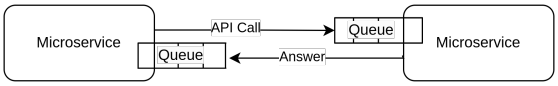

Figure 3: Queue Pattern

Using a queue enables asynchronous message processing without blocking. The queue receives all incoming messages and stores them. If the Service wants to process one request, it can get the next item in the queue. Additionally, the Competing Consumers Pattern can be used to have multiple consumers[28]. This pattern can also be used inside an api gateway.
Richardson et al.[26] mentions two different styles: Notification (without a reply) and Request/asynchronous response.
In Fig:3 the request asynchronous variant is shown.

3.3.1 Advantages

This pattern scales very well in all three axes[24]. It helps to minimize costs through scaling down the number of consumers when no demand is there[28]. Then it allows scaling the throughput by scaling up the number of consumers[28]. Using a queue makes batch processing of similar requests, which is more performant possible[2]. Queues increase resilience because messages are stored when they cannot be processed at the moment[13]. Using a priority Queue allows prioritizing of some customers[28].

3.3.2 Disadvantages

No Disadvantages were found.

3.3.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 4: Queue Quality Attributes

3.4 API Gateway Pattern

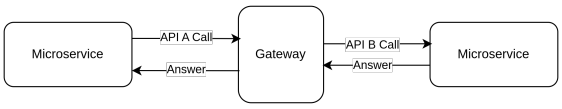

Figure 4: API Gateway Pattern

The requesting service calls a gateway service that forwards the request to the right recipient.
Common implementations used are the RabbitMQ Broker[7], ActiveMQ[8] or Apache Kafka[9]. Hong et al. [12] performance tested rabbit mq against and found RabbitMQ slower with a low number of messages (less than 250) but faster and more stable with more messages. Most of the following patterns require the use of this pattern and can be seen as sub patterns of it.

3.4.1 Advantages

Through introducing a middle service, the communicating serves do not have to know each other, promoting encapsulation[24][37, p.180f]. This allows easy to extend functionality, backwards compatibility[32] and improves Security[36][21, p.97f]. In case of network failures, messages can be short time stored, promoting resilience[37, p.180f]. The pattern allows easy scaling[22] and reduces the number of network calls[24]. Gateways can accept messages immediately and deliver them later, so services do not block longer than necessary[37, p.180f].

3.4.2 Disadvantages

Introduces a single point of failure[16] that can be a potential bottleneck[32]. The extra Services need added Maintenance and finding bugs in a bigger system is more complex[24].

3.4.3 Quality Attributes

[7] https://www.rabbitmq.com/
[8] https://activemq.apache.org/
[9] https://kafka.apache.org/

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 5: API Gateway Quality Attributes

3.5 Anti-Corruption Layer (Adapter) Pattern

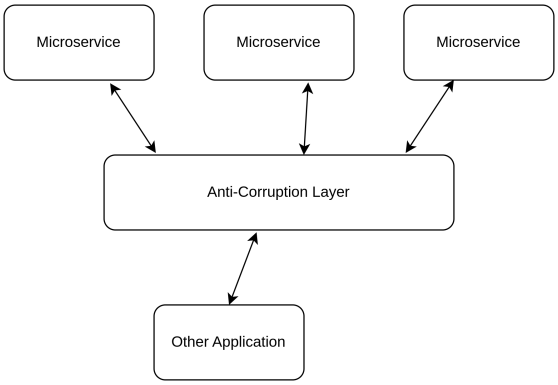

Figure 5: Anti-Corruption Pattern

Especially when migrating a large system to a microservice architecture, parts of the old system may still exist. Even when fully migrated, some parts of the system may remain as they were because a transition is too expensive or complicated. In this case, an anti-corruption Layer can be placed between different systems. It hides application implementation details by translating requests, thus providing a simple interface to the other systems.

3.5.1 Advantages

Enables communication between services with different semantics[28][4, p.22f]. Also, enable communication with services not prior part of the system without modifying them[5].

3.5.2 Disadvantages

Adds additional Latency and an extra Service to manage[28]. Scaling this service can be hard[28].

3.5.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 6: Anti Corruption Layer Quality Attributes

3.6 Aggregator Pattern

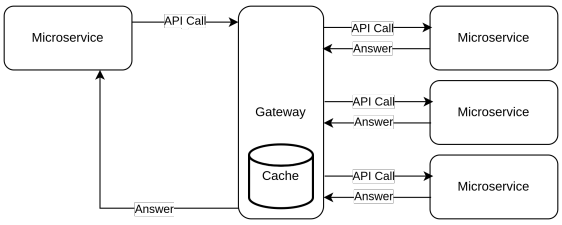

Figure 6: Aggregator Pattern

If a service needs data from multiple sources, a special aggregator service can be used that collects information from multiple different other services. An Aggregator can be used as a client entrypoint. It can have a database in which it can cache gathered information.

3.6.1 Advantages

Reduces the number of requests, thus improves latency, especially on high-latency networks[28]. Introduces a higher degree of encapsulation[24].

3.6.2 Disadvantages

Introduce a single point of failure and a potential bottleneck[28]. Latency is introduced by the additional service[24, 36].

3.6.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 7: Aggregator Pattern Quality Attributes

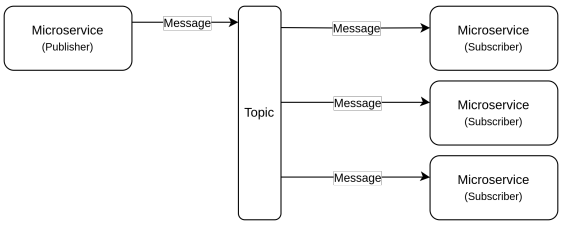

Figure 7: Publish/Subscribe communication Pattern

3.7 Publish/Subscribe communication Pattern

Messages are published on topics. Message Consumers can subscribe to topics to receive all published messages.
Apache Kafka[10] is a popular implementation of this pattern.

3.7.1 Advantages

Straightforward to add new listeners[9][37, p.180f] due to high decoupling[28]. Message receivers can decide which messages they want to receive[28]. Topics can be monitored to find problems easily[28].

3.7.2 Disadvantages

Not suitable for realtime communication or when only a small number of services participate in the communication[28].

3.7.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 8: Publish/Subscribe Quality Attributes

3.8 Ambassador Pattern

Every microservice has to communicate with other microservices. Because of that, it can make sense to offload this common functionality to a helper service that does the communication for a microservice. The Ambassador can handle tasks like encryption, authentication, routing, metering, monitoring and logging, and other communication patterns. If only parts of the messaging functionality are offloaded, this pattern is called **Gateway Offloading**.

[10] `https://kafka.apache.org/`

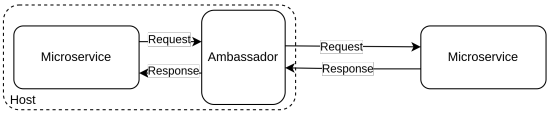

Figure 8: Ambassador Pattern

3.8.1 Advantages

Specialized teams can develop Common communication features[28]. It Is an easy way to support multiple languages or legacy applications[28]. The same code can be reused across many languages and applications[5].

3.8.2 Disadvantages

Adds latency to the communication process that does not exist when using a library[28]. Can introduce a bottleneck if it is not scaling[28]. If the ambassador handles retries, data consistencies have to be considered[28]. May not be suitable when the communication needs to be deeply integrated into the microservice[28].

3.8.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 9: Ambassador Pattern Quality Attributes

3.9 Backends for Frontends (BFF) Pattern

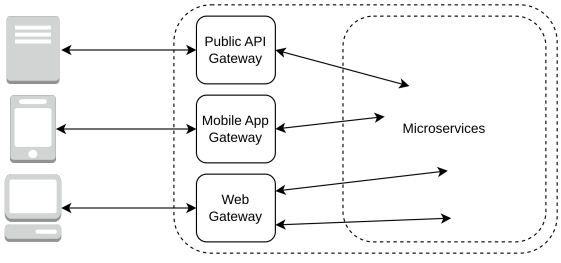

Figure 9: Backends for Frontends Pattern

For every different client type accessing the product, there is a specific backend gateway that handles the request. The different backend gateways work independently of each other.

Newman [22] describes two different variations. There can be strict separation between clients like IOS Application and Android Application, each having their own gateway or each ”experience”, like mobile devices have their own gateway. They observe that in the industry this decision depends on internal team structures.

3.9.1 Advantages

Reduces development overhead from a complicated general purpose backend[28]. A less complicated back end entrance improves security[36]. Optimizes requests from specific clients, thus reducing latency[28, 36][37, p.174]. Backend Gateways can be owned by the same team developing the frontend, making release faster and simplifying communication[22][37, p.175].

3.9.2 Disadvantages

A lot of clients use the same basic functionality that has to be implemented more than once. Code duplikation is likely[28].

3.9.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 10: Backends for Frontends Quality Attributes

3.10 Gateway Routing Pattern

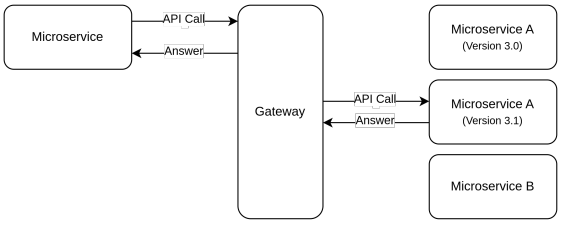

Figure 10: Gateway Routing Pattern

The Gateway service in this pattern can analyze route incoming requests and route them to different targets.
Robbag et al. [28] identifies three different types of targets. Multiple instances of the same service to implement load balancing. Multiple versions of the same service to upgrade

only some services or continue to support legacy clients. Multiple different services to allow changes in the structure without affecting other parts.
This pattern can also function as an entrypoint for clients.
If requests are routed between different versions, this pattern is called **Canary Pattern/StrangleFit Pattern)**.

3.10.1 Advantages

Simplifies deployment and routing for more complex systems[28].

3.10.2 Disadvantages

Introduces a single point of failure and a potential bottleneck[28].

3.10.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 11: Gateway Routing Quality Attributes

3.11 Chained and Branching Microservice Communication Pattern

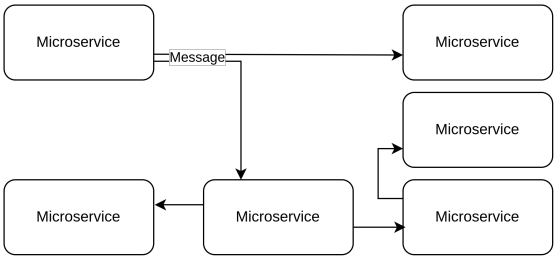

Figure 11: Chained and Branching Microservice Communication Pattern

If a service needs information from multiple services, it can get them by dynamically calling the right ones. The calls can be multiple layers deep. Calls can go to multiple services at the same time.

3.11.1 Advantages

Very flexible implementation, good scalability[24].

3.11.2 Disadvantages

With long chains of communication, the latency can become very high[24].

3.11.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 12: Chained and Branching Communication Quality Attributes

3.12 Circuit Breaker Pattern

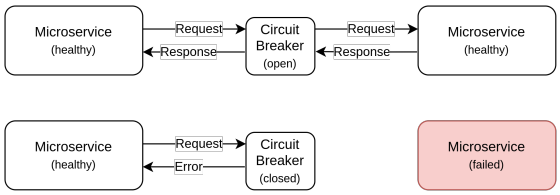

Figure 12: Circuit Breaker Pattern

When a messaging operation fails, it needs to be retried to perform the operation. But continuous retries from multiple services can overload the recovering service. It also ties up resources on the trying service. This can lead to a cascading failure if the trying service gets a request. The Circuit Breaker Pattern stops the retries after a certain number of attempts and has predefined error handling.
Montesi et al. [20] observe three different forms of Circuit Breakers: Client-side Circuit Breaker, Server-side Circuit Breaker, and Proxy Circuit Breaker as shown in 12.
Montesi et al. [19] and Microsoft [28] also introduce a state machine design that consists of three states: Open, Half-Open, and Closed. In the open state all requests go through. If in a predefined time too many requests fail, the circuit breaker goes into the Closed State. In this state requests fail immediately. After a predefined time the state changes to the half-open state. In this state some requests are allowed through. If they fail, the state changes back to closed, if they succeed, it changes to the open state.

3.12.1 Advantages

Improves reliability when accessing services that are likely to fail[28]. Less resource usage in the client than using a naive retry method[18].

3.12.2 Disadvantages

Every request has to go through this mechanismus so some latency and overhead are added[17, 28].

3.12.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 13: Circuit Breaker Quality Attributes

3.13 Time-Outs Pattern

A Service should not wait for a reply forever because waiting blocks resources. After a predefined time it should give up. Time-Outs can be applied to all communication, operations, and events in a microservice system.

3.13.1 Advantages

TimeOuts stopp malfunctioning services from affecting other services[22, p.397ff]. By logging Time-Outs slow systems can be identified[22, p.397ff].

3.13.2 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 14: Time-Outs Quality Attributes

3.14 Retries Pattern

If an Operation fails, it is repeated after some time.
Robbag et al.[28] describes two variants: Retry immediately if the error code is rare and Retry after Delay if the error code implies a network or load issue.

3.14.1 Advantages

Retrying improves the availability of services when they are temporarily overloaded[22, p.399f][18].

3.14.2 Disadvantages

Too many retries too fast can overwhelm the already struggling service[22, p.399f][28]. Using this pattern can mask scaling issues. If services are frequently overloaded, they should be scaled up[28]. Retries can have cause operations to be performed more than once[28]. The retried message needs to lead to an idempotent operation.

3.14.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 15: Retries Quality Attributes

4 Deployment Architecture Patterns Discussion

4.1 Serverless Deployment Pattern

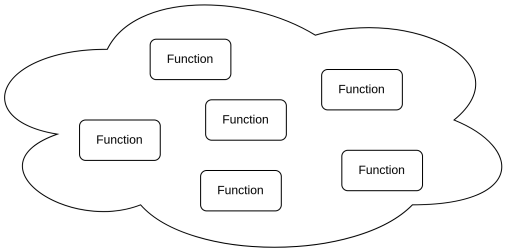

Figure 13: Serverless Deployment Pattern

On Serverless deployment infrastructure, small stateless components can be deployed that get invoked by events. These components are called functions and are automatically scaled to handle the existing load. The cost is dependent on usage.
AWS Lambda[11], Google Cloud Functions[12], Vercel[13] and Azure Functions[14] are the most used Serverless Cloud providers.
Newmann[22, p.253ff] shows three different approaches for dividing microservices into functions: Function per Service, Function per aggregate, and fine-grained. Function per aggregates means dividing one microservice into the different objects families it handles. Each object familie (aggregate) then gets handled by a function. Dividing functions even smaller is also possible but has the risk of creating data inconsistency when one object/event is split in different functions[22, p.255f].

4.1.1 Advantages

This type of deployment in general scales very well and is cheaper than managing own infrastructure[26]. In some specific workloads using this pattern can be slower and more

[11] https://aws.amazon.com/lambda/
[12] https://cloud.google.com/functions/?hl=en
[13] https://vercel.com/
[14] https://azure.microsoft.com/en-us/products/functions/

expensive[14, p.9ff]. This pattern reduced development complexity because there is no need to manage infrastructure[11, p.10][26]. The big Cloud providers providing this service can deploy close to the end user, so latency is reduced[11, p.10].

4.1.2 Disadvantages

Some cloud providers enforce a maximum run time per function, and most functions are stateless[22, p.251]. This limits what functionality that can be developed with this pattern. Developers also risk Vendor Lock in[11, p.16f]. The powerful automatic scaling can overwhelm other parts of the system if the rest of the system does not scale equally fast[p.252][22]. This autoscaling can also lead to high, unpredictable costs. Another downside is especially with big functions, startup times, and thus latency can be high[26][11, p.16].

4.1.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 16: Serverless Deployment Quality Attributes

4.2 Service Deployment Plattform Pattern

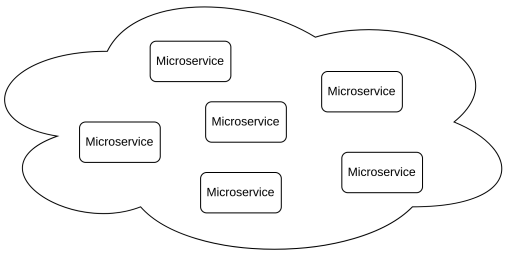

Figure 14: Service Deployment Plattform Pattern

The service deployment plattform provides an automated low-level infrastructure, so the developer just needs to build the microservice.
The Service Instance per Container4.3 is mostly used inside this pattern as a way to package software that should be deployed.
AWS Fargate[15], Heroku[16] and CloudFoundry[17] are often used cloud providers.

[15] https://aws.amazon.com/fargate/
[16] https://www.heroku.com/
[17] https://www.cloudfoundry.org/

4.2.1 Advantages

Service Deployment Platforms provide automatic scaling, so increases in demand are automatically met[11, p.9]. They can also scale down on low-demand reducing cost[11, p.9]. Cost can be reduced in development too. Because there is no need to manage infrastructure, development and maintainability complexity is reduced[11, p.10]. Like with Serverless Deployments, the big cloud providers have servers close to the end user, reducing latency[11, p.10].

4.2.2 Disadvantages

Similar to the serverless deployment pattern, this pattern introduces vendor lock in[11, p.16f] and limits functionality to provided use cases[22]. The latency when no services are currently running is high (Cold Start)[11, p.16].

4.2.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 17: Service Deployment Plattform Quality Attributes

4.3 Service Instance per Container Pattern

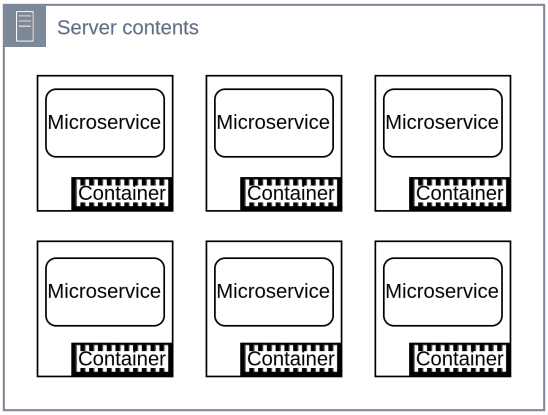

Figure 15: Service Instance per Container Pattern

With this pattern there are multiple containers running on one physical host. Each container runs one microservice. A Container is a standalone, executable package that includes everything needed to run a piece of software, including the code, a runtime,

libraries, environment variables, and config files. Processes running inside a container are isolated from processes in other containers but share the host kernel.
Software like Docker Containers[18], LXC[19] or LXD[20] are used to deploy containers.
Software like Kubernetes[21] or Docker Swarm[22] can be used to orchestrate and manage multiple containers on multiple hosts.

4.3.1 Advantages

In contrast to Virtual Machines (VM), containers introduce very little overhead compared to bare metal performance[22, 30]. By using container orchestrators, a container system can automatically scale to load[22, p.259][26]. Container images are smaller than VM images. They can be developed faster, speeding up development[26]. They also start up faster, and resource limitations can be put in place[26]. Docker Containers are designed to run only one process inside them, complying with the microservice principle[21, p.93f].

4.3.2 Disadvantages

No disadvantages found in the literature.

4.3.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 18: Service Instance per Container Quality Attributes

4.4 Service Instance per Virtual Machine Pattern

With this pattern there are multiple Virtual Machines (VM) on every physical host. Each VM only contains on microservice service.
Software like VmWare vSphere Hypervisor[23], KVM[24], Quemu[25] or Hyper-V[26] is used to deploy VMs.

[18] https://www.docker.com/resources/what-container/
[19] https://github.com/lxc/lxc
[20] https://canonical.com/lxd
[21] https://kubernetes.io/
[22] https://docs.docker.com/engine/swarm/
[23] https://www.vmware.com/products/vsphere-hypervisor.html/products/vsphere-hypervisor.html
[24] https://www.linux-kvm.org/page/Main_Page
[25] https://www.qemu.org/
[26] https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

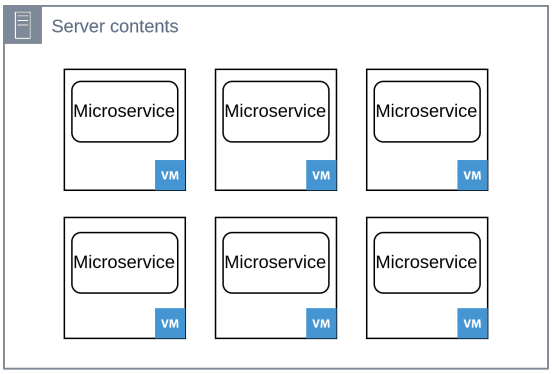

Figure 16: Service Instance per VM Pattern

4.4.1 Advantages

VMs offer high isolation down to the kernel. This allows scaling by duplikation and limitation of resources[26].

4.4.2 Disadvantages

VM images are larger than containers and because of their bigger isolation introduce more overhead[22]. This leads to worse performance than using containers in all scenarios[23, 15].

4.4.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 19: Service Instance per VM Quality Attributes

4.5 Service Instance per Host Pattern

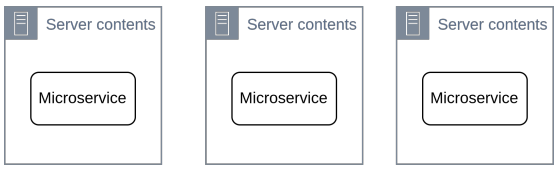

Figure 17: Service instance per Host Pattern

Every physical host runs only one service. Basically the same as 4.4 but on independent hardware.
Few papers and articles can be found describing or using this pattern which shows that the pattern is not widely used.

4.5.1 Advantages

Different physical hosts offer the highest degree of isolation possible[32].

4.5.2 Disadvantages

Scaling physical hardware fast is hard[32], and hardware is often not fully used and thus wasted[22, 26].

4.5.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 20: Service Instance per Host Quality Attributes

4.6 Multiple Service Instances per Host Pattern

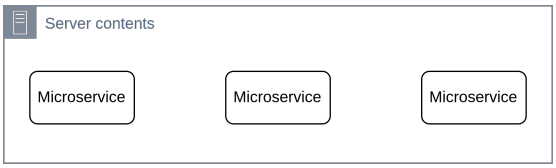

Figure 18: Multiple Service Instances per Host Pattern

On every physical host, there are multiple services running without more separation than the Operating System provides.
Few papers and articles can be found describing or using this pattern which shows that the pattern is not widely used.

4.6.1 Advantages

This pattern provides bare metal performance[32].

4.6.2 Disadvantages

This pattern violates the basic principle of isolation[22] and independence used to define microservices. Resulting, monitoring, or limiting services independently is hard[26]. Processes on the same host use shared Libraries, which can cause dependency conflicts[26].

4.6.3 Quality Attributes

Scalability	Performance	Availability	Monitorability
Security	Testability	Maintainability	Cost optimization

Table 21: Multiple Service Instances per Host Quality Attributes

5 Conclusion

We identified six deployment patterns and twelve communication patterns for microservice applications. The patterns came from 17 academic papers, five textbooks, six academic literature reviews, three industrie reports and two industrie learning materials. In Fig: 19 the result of the classification can be seen using a tree style division. Additionally, all patterns were examined about their impact on eight quality attributes. All patterns in the communication categories provide surplus value and have use cases in different scenarios. This is not the case with the identified deployment patterns. While Service Instance per VM has some use cases, Service Instance per Host and Multiple Service Instances per Host can be categorized as microservice anti-patterns. Service Instance per Container is by far the most-used pattern, it can be argued that this pattern defines microservices and makes them possible.
When using a large microservice system, deploying additional communication gateway services is a good idea. If performance and latency matter a lot or few services are involved, the communication can be handled by the services themselves.

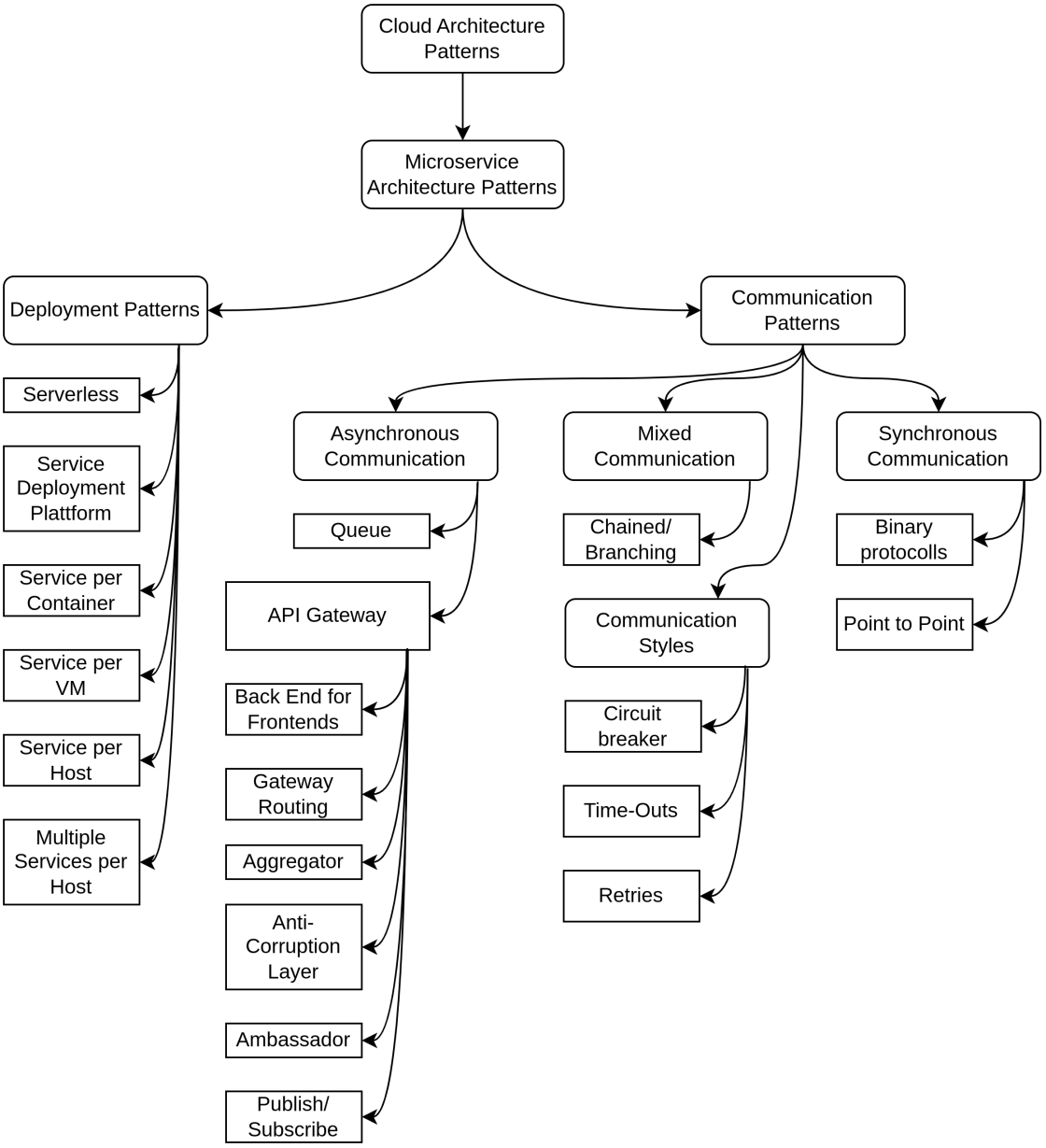

Figure 19: Tree of Patterns

References

[1] Martin Nally (Google). *gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design*. https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them. [Online; accessed 12-Jun-2024]. 2020.

[2] Akhan Akbulut and Harry G Perros. "Performance analysis of microservice design patterns". In: *IEEE Internet Computing* 23.6 (2019), pp. 19–27.

[3] Deval Bhamare et al. "Exploring microservices for enhancing internet QoS". In: *Transactions on Emerging Telecommunications Technologies* 29.11 (2018), e3445.

[4] Kyle Brown and Bobby Woolf. "Implementation patterns for microservices architectures". In: *Proceedings of the 23rd conference on pattern languages of programs*. 2016, pp. 1–35.

[5] Brendan (Google) Burns and David (Google) Oppenheimer. "Design patterns for container-based distributed systems". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016. URL: https://storage.googleapis.com/gweb-research2023-media/pubtools/pdf/45406.pdf.

[6] Björn Butzin, Frank Golatowski, and Dirk Timmermann. "Microservices approach for the internet of things". In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–6.

[7] John Carnell and Illary Huaylupo Sánchez. *Spring microservices in action*. Simon and Schuster, 2021. URL: https://annas-archive.gs/md5/3c6dc613426280ca3eafe43dea24391e.

[8] Tomas Cerny et al. "Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study". In: *Journal of Systems and Software* 206 (2023), p. 111829. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111829.

[9] Namiot Dmitry and Sneps-Sneppe Manfred. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27.

[10] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering* (2017), pp. 195–216. URL: https://arxiv.org/pdf/1606.04036.

[11] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. "Survey on serverless computing". In: *Journal of Cloud Computing* 10 (2021), pp. 1–29. URL: https://link.springer.com/article/10.1186/s13677-021-00253-7.

[12] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. "Performance analysis of RESTful API and RabbitMQ for microservice web application". In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE. 2018, pp. 257–259.

[13] IBM. *What is a message queue?* https://www.ibm.com/topics/message-queues. [Online; accessed 14-Jun-2024]. 2020.

[14] Eric Jonas et al. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019). URL: https://arxiv.org/pdf/1902.03383.

[15] Ann Mary Joy. “Performance comparison between Linux containers and virtual machines”. In: *2015 international conference on advances in computer engineering and applications*. IEEE. 2015, pp. 342–346.

[16] Işıl Karabey Aksakalli et al. “Deployment and communication patterns in microservice architectures: A systematic literature review”. In: *Journal of Systems and Software* 180 (2021), p. 111014. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.111014. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001114.

[17] Shanshan Li et al. “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review”. In: *Information and software technology* 131 (2021), p. 106449.

[18] Nabor C Mendonca et al. “Model-based analysis of microservice resiliency patterns”. In: *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE. 2020, pp. 114–124. URL: doi.org/10.1109/ICSA47634.2020.00019.

[19] Fabrizio Montesi and Janine Weber. “Circuit breakers, discovery, and API gateways in microservices”. In: *arXiv preprint arXiv:1609.05830* (2016).

[20] Fabrizio Montesi and Janine Weber. “From the decorator pattern to circuit breakers in microservices”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1733–1735.

[21] Irakli Nadareishvili et al. *Microservice architecture: aligning principles, practices, and culture*. " O’Reilly Media, Inc.", 2016. URL: https://www.programmer-books.com/wp-content/uploads/2019/07/Microservice-Architecture.pdf.

[22] Sam Newman. *Building microservices (2.Edition)*. "O’Reilly Media, Inc.", 2021. URL: https://annas-archive.gs/md5/01c3abc044d7bd3aac5dac4c9c95c545.

[23] Amit M Potdar et al. “Performance evaluation of docker container and virtual machine”. In: *Procedia Computer Science* 171 (2020), pp. 1419–1428.

[24] Mahir Ramchand. “A Systematic Mapping of Microservice Patterns”. PhD thesis. 2021. URL: https://fse.studenttheses.ub.rug.nl/25671/.

[25] Mark Richards. *Software Architecture Patterns, Second Edition*. O’Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA, 2022. ISBN: 9781098134273. URL: https://annas-archive.gs/md5/b1159e8fb0da957920467b5db8a9227f.

[26] C. Richardson. *Microservices Pattern: A pattern language for microservices*. http://microservices.io/patterns/. [Online; accessed 13-Mai-2024]. 2024.

[27] C. Richardson. *Microservices patterns: with examples in Java*. Manning Publications, 2019. ISBN: 978-1-61729-454-9. URL: https://annas-archive.gs/md5/1d08603a7452d4496fdb64f05c35b784.

[28] robbag. *Cloud Design Patterns*. https://learn.microsoft.com/en-us/azure/architecture/patterns/. [Online; accessed 13-Mai-2024]. 2024.

[29] Mark Russinovich et al. *Microsoft Azure Well-Architected Framework pillars*. https://learn.microsoft.com/en-us/azure/well-architected/pillars. [Online; accessed 10-Jun-2024]. 2023.

[30] Prateek Sharma et al. "Containers and virtual machines at scale: A comparative study". In: *Proceedings of the 17th international middleware conference*. 2016, pp. 1–13.

[31] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. "Thrift: Scalable cross-language services implementation". In: *Facebook white paper* 5.8 (2007), p. 127. URL: https://thrift.apache.org/static/files/thrift-20070401.pdf.

[32] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. "Architectural patterns for microservices: a systematic mapping study". In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress. 2018.

[33] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[34] J. A. Valdivia et al. "Patterns Related to Microservice Architecture: a Multivocal Literature Review". In: *Programming and Computer Software* 46.8 (Dec. 2020), pp. 594–608. ISSN: 1608-3261. DOI: 10.1134/S0361768820080253. URL: https://doi.org/10.1134/S0361768820080253.

[35] Guilherme Vale. "Designing Microservice Systems Using Patterns: An Empirical Study On Architectural Trade-offs". en. In: (2021).

[36] Muhammad Waseem et al. "Decision models for selecting patterns and strategies in microservices systems and their evaluation by practitioners". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 2022, pp. 135–144. URL: https://doi.org/10.1145/3510457.3513079.

[37] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, 2016. URL: https://annas-archive.gs/md5/567bf83cd7c75093419f6f2fb08fa998.

If one of the used Websites should not be online anymore, snapshots of most sources can be found at https://archive.org/.

AI Usage in this thesis:

- Languagetool.org (Spellcheck, Grammar check, and some sentence Rephrasing)
- JetBrains Grazie (Spellcheck and Grammar check)
- GitHub Copilot (Latex Commands autocomplete, some sentence Rephrasing, very minor sentence autocompletion)
- DeepL (Translations)