jamk

Serverless and microservice architecture in modern software development

Jussi Heikkinen

Masters thesis
April 2023
Information technology
Full Stack Software Development

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Heikkinen, Jussi

Serverless and microservice architecture in modern software development

Jyväskylä: Jyväskylän ammattikorkeakoulu. Huhtikuu 2023, 47 sivua.

Informaatioteknologia, Full Stack Software Development. Opinnäytetyö YAMK.

Julkaisun kieli: Englanti

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

Ohjelmistokehitys on ympäristö, joka on jatkuvassa muutoksessa ja jossa teknologiat ja toimintatavat kehittyvät nopeaa vauhtia. Sovelluksia on rakennettu monoliittisella arkkitehtuurilla jo useita vuosikymmeniä ja niiden ylläpidettävyys, päivitettävyys ja skaalautuvuus on alkanut saavuttaa rajansa. Mikropalveluarkkitehtuuri yrittää ratkaista monoliittisten järjestelmien ongelmia lisäämällä niiden modulaarisuutta, sekä lisäämällä uuden ulottuvuuden järjestelmien skaalautumiseen. Tästä syystä päätin tutkia miten mikropalveluarkkitehtuuri ja palveliton arkkitehtuuri ovat parantaneet ohjelmistojen laatua ja kehittäjien kokemusta ohjelmistokehitystyöstä.

Opinnäytetyön tavoite oli teettää tutkimus, jossa kerättiin dataa ohjelmistokehityksen ammattilaisilta ja verrata tuloksia aiempaan tutkimustietoon. Tuloksia vertaamalla voitiin tarkastella, onko mikropalveluarkkitehtuuria osattu hyödyntää oikein ja kuinka se on vaikuttanut lopputulokseen

Määrälliset ja laadulliset tulokset tutkimuksesta osoittavat, että mikropalvelu ja palveliton arkkitehtuuri ovat ratkaisseet ongelmia, joita ei voitu ratkaista monoliittisissa järjestelmissä, ja että kehittäjäkokemus on pysynyt ennallaan tai jopa parantunut siitä huolimatta, että järjestelmien kompleksisuus ja granulariteetti on kasvanut huomattavasti.

Avainsanat (asiasanat)

Microservices, Serverless, AWS, Azure, Domain Driven Design, Modern software development

Muut tiedot (salassa pidettävät liitteet)

Työ ei sisällä salassa pidettävää materiaalia

jamk

Description

Heikkinen Jussi

Serverless and microservice architecture in modern software development

Jyväskylä: JAMK University of Applied Sciences, April 2023, 47 pages.

Information Technology, Full Stack Software Development. Masters thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

Software development is a constantly evolving space where technologies and ways of work change rapidly. Applications have been built as monoliths for decades and they have reached their limit in terms of maintainability, upgradeability, and scalability. Microservice architecture tries to tackle all their problems and allow a more modular approach to software development allowing new dimension to scalability compared to what was possible in monolithic systems. The downside of microservice architecture is that it introduces an elevated level of complexity and thus requires longer time to deliver a product compared to monolithic architecture. For this reason, I decided to study how microservices and serverless architecture have affected software quality and developing experience for developers.

Objective for research was to conduct a survey to gather data from experts and compare results to previous research. By comparing the results, the target was to see if possibilities of the microservice architecture have been successfully taken into use and how it has affected the developing experience.

Results from qualitative and quantitative data suggests that microservice architecture and serverless architecture have been able to solve some of the problems that were not possible in monolithic systems and that the developing experience have mostly stayed same or increased regardless the added complexity of the distributed systems.

Conclusion is that microservice architecture is a promising path for modern software development and it is a key to solving the issues with scalability and performance compared to monolithic systems built in the past.

Keywords/tags (subjects)

Microservices, Serverless, AWS, Azure, Domain Driven Design, Modern software development

Miscellaneous (Confidential information)

This research does not contain confidential information

Contents

1 Preface

Monolithic architecture has been a traditional approach to develop applications for decades. In recent years the use of web services has increased a lot and these monolithic systems have faced their limits in scalability and maintainability. *To cope with the constant and fast-changing business needs, it is a must for an organisation to move from a monolith to a microservices architecture to bring changes quickly, innovate fast, be ahead in the competition, reduce time-to-market timelines* (Khaitan Nitin). The technologies are getting old and it's difficult-to-find developers who want to write code for these systems. There are many problematic areas in monolithic architecture like tight coupling and long release cycles. Code might be hard to understand as the size of codebase increases. Working on large codebase requires *different ceremonies like deployment through environments, functional testing, load testing, security testing, integration testing, user acceptance, etc* (Khaitan Nitin). Working on a codebase like this is a burden for developers.

As the demand for high quality software increases the new quality standards and nonfunctional requirements for software have emerged. Microservices offer a solid approach to fulfill these requirements and to solve the problems of monolithic systems by enabling new tools to scalability, modularity and developing experience.

2 Purpose, objectives and the research methods

The purpose of this research is to clarify how microservice architecture and serverless architecture improve quality of software. How developers and end users benefit from using microservice architecture and which technologies have been established in modern software developing conventions. The objective is to find out how new extent of architectural choices, modern technologies, cloud platforms and decomposition of business domain affects the development cycle, maintainability, reliability, and performance of the software.

The research is made using triangulation which as described by Pritha Bhandari *means addressing research question by multiple datasets, methods, theories, or investigators. Using triangulation research strategy enhances the validity and credibility of findings.* Action research method is used to analyze existing research of the topic and quantitative and qualitative data is collected using survey for a small scope of experts. Objective is to declare the current state of the micro-service

architecture and how it has redeemed its goals and how experts find it solving the problems it is designed.

3 Architecture concepts

3.1 Microservice architecture

3.1.1 Introduction

Microservice architecture is vastly different to Monolithic architecture where a single codebase is layered using abstraction and modules. *Microservices are small autonomous services that work together (Sam Newman, 2).* Appropriately built microservices fulfill these two rules. It must have loose coupling to other microservices, and it must be built with focusing on high cohesion. *When services are loosely coupled, a change to one service should not require a change to another. The whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system. (Sam Newman, 30.)* Loose coupling provides developers confidence to make rapid changes without affecting other services and the stability of the entire system. High cohesion is achieved by collecting related behavior and models together that change for the same reason.

Each microservice is preferably developed by a small team who writes code, tests, and handles the deployment. They have freedom to choose the best suited technologies for the microservice they are developing because services communicate over interfaces usually using HTTP -protocol. For some services where performance is crucial a low-level programming language might be the best choice bundled with performant database solution and where performance is secondary a high-level language familiar to team can be sufficient.

3.1.2 Decomposing business domain

While developing a new system or decomposing and monolithic software to microservices there is similar paths to reach the targets. *Decomposing by business capability is a strategy for creating a microservice architecture. The essence is to capture things that the business does in order to generate value as business capabilities. (Chris Richardson, 51.)* This decomposing strategy focuses on defining what the business is instead of defining how the business works to achieve its goals.

Another strategy which is getting vastly popular is decomposing by subdomains. It is based on a book Domain-Driven Design by Eric Evans. *The idea is that any given domain consists of multiple bounded contexts, and residing within each are things that do not need to be communicated outside as well as things that are shared externally with other bounded contexts. Each bounded context has an explicit interface, where it decides what models to share with other contexts. (Sam Newman, 31.)* Domain driven design does give tools to design and develop microservice architecture. It does not give a guideline or finished steps to follow by. An architect must read and adapt the theory with the working team to find their way to define the bounded contexts and service boundaries.

Domain driven design consists of two parts. Strategic phase and tactical phase. Strategic DDD aims to define the business domain from a high perspective. It divides business domain, the so-called problem space of the whole application to smaller areas called subdomains which each have their own functional problem they will be solving. Tactical DDD will continue from strategic phase and aims to define the modeling of things and system operations in the subdomain. Microservices will be built upon this modeling.

3.1.3 Ubiquituous language

Ubiquitous language is essential to a team working on a project. As domain experts and developers communicate about the functionality and modeling of the system, they might use their own business language and they might have problems understanding what the other party is trying to say.

As we use the ubiquitous language of the domain model in discussions, especially discussions in which developers and domain experts hash out scenarios and requirements, we become more fluent in the language and teach each other its nuances. We naturally come to share the language that we speak in a way that never happens with diagrams and documents. (Eric Evans, 19.)

According to Eric Evans (2001, 19.), when you talk about the system, you must be willing to play with the model. Think about the concepts allowed by the model and describe its scenarios aloud using its interactions and elements. Work around your diagrams and code and find an easier way to communicate the ideas what they need to say.

From these discussions the team should gather the ubiquitous language and write it down. Eric Evans describes and intersection of two circles and in the intersection is all the vocabulary the team uses in discussion, documentation, and code. If something changes in the language it is reflected in all the documentation and code. The circle has two outer sections of which another has the technical terminology that is not used in modeling the system and the other have business terms developers do not understand and business terms that are used but are not used to modeling the software. These terms are merged into the design later if they start to appear often and the design does not have simpler way to describe them.

3.1.4 Defining the bounded contexts

Bounded contexts are the subdomains and their implementation in the applications business domain or problem space. Each subdomain defines an explicit context for the methodology and concepts. *In an ideal world, we would have a single model spanning the whole domain of the enterprise. This model would be unified, without any contradictory or overlapping definitions of terms. Every logical statement about the domain would be consistent. (Eric Evans, 214.)* In large enterprise applications we must allow the same model to develop differently in various parts of the application scope. Developers do not need to agree on one unified model for the problem space that can be increasingly complex to a simple task. Instead, they can choose to create a similarly named model or multiple models to describe the same thing in different bounded contexts. Usually, it is good to have a context map to describe the mapping of models in the domain.

Cells can exist because their membranes define what is in and out and determine what can pass (Eric Evans, 216). This behavior from biology is usable in microservice architecture. A bounded context can consist of one or several microservices. These microservices have interfaces they use to share data and functionality inside the bounded context. The bounded context has an explicit interface used to communicate with other bounded contexts. The explicit interfaces use shared models to translate information between bounded contexts. In some scenarios where an entity belongs to two bounded contexts and the developer would need to duplicate business logic it is possible to use a shared kernel, an overlapping of two bounded context.

3.1.5 Unraveling the bounded contexts

In domain-driven design's strategic phase you are mapping out the business domain and extracting the bounded contexts for your domain models. Domain models are defined with further accuracy in tactical domain-driven design phase. Each bounded contexts are applied with the tactical patterns (AZURE TACTICAL DDD). There are a few patterns of model element that express the model for domain driven design. Each pattern defines characteristics of a class in a domain model. Entities, value objects, modules and services express models in a software and aggregates use them to encapsulate objects in a software. Factories and repositories complete the design providing the outbound adapter to data sources and initialization of the objects. These patterns are called model driven design. Model driven design helps us understand the role of model in the domain, its associations, and what objectives it has.

3.1.6 Entity

An object defined primary by its identity is an entity (Eric Evans, 55). Entities are persistent and cannot be replaced by any other entity. Entity has an identity that can be tracked in a system, or it can span over multiple systems like social security number. Two entities with the same attributes are not the same if the identifiers do not match. In distributed systems developers can generate a universally unique identifier to objects that use the entity model pattern.

3.1.7 Value object

An object that represents a descriptive aspect of the domain with no conceptual identity is called a value object. Value objects are instantiated to represent elements of the design that we care about only for what they are, not who or which they are. (Eric Evans, 59.) A value object is something that has no identity. Value object is preferably immutable and only its attributes are important to developer. A real-world example of a value object would be a chair around a table. It does not matter which chair I used last time and there is no way of telling it afterwards. We are only interested in how it can be used to sit on and that is how it brings value.

3.1.8 Service

A service is an operation offered as an interface that stands alone in the model, without encapsulating state, as entities and value objects do. Services are a common pattern in technical frameworks, but they can also apply in the domain layer. (Eric Evans, 63.) Services are objects that hold logic to action or activities. This logic is separated to a stand-alone object to keep other entities simple and clear. Services are the doers of the applications, and they are named after activities, for example orchestrator or a messenger. Services are stateless and they can be used by any entity or value object.

3.1.9 Module

Modules have a small role in microservices, but they are part of model driven design. Modules are a way to put together several classes and give them one understandable name and purpose. Modules enhance readability and low coupling of software.

Like everything else in a domain-driven design, modules are a communications mechanism. The meaning of the objects being partitioned needs to drive the choice of modules. When you place some classes together in a module, you are telling the next developer who looks at your design to think about them together. If your model is telling a story, the modules are chapters. The name of the module conveys its meaning. (Eric Evans, 67.)

3.1.10 Aggregate

According to Eric Evans (2001, 76.), An aggregate is a unit that we treat as a purpose for data changes forming a cluster consisting of associated objects. Each aggregate has a root, a single and specific entity and a boundary defining what belongs to inside the aggregate. Objects outside aggregate are only allowed to hold references to root entity of aggregate and objects within the boundary are only allowed to hold references to each other. Other entities than the root entity in aggregate is not visible to objects outside of the aggregate context and thus have a local identity that is distinguishable only within the aggregate.

Identifying aggregates is particularly important because microservices are formed of one or more aggregates. Aggregate has business rules called invariants. Every change in aggregate boundary must fulfill all the invariants to complete. Entities and value objects within aggregate have only local identity and they serve no purpose without relation to a root entity so they must be removed with it.

3.1.11 Factory

When an initialization of object is complicated, and the object being created has no need for the initialization logic it makes sense to shift the responsibility to a factory class. Factory classes are the assemblers of objects. We can think of a farmer as a factory object. Farmer plants the seeds and takes care of the plants but when the plants are harvested, they have no use for information of how they were created, and they will not need to reuse that logic.

Shift the responsibility for creating instances of complex objects and aggregates to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the objects being instantiated. (Eric Evans, 84.)

3.1.12 Repository

A repository is an object that hides the implementation of persistent storage. If developers have access to databases by making database queries anywhere in code, it compromises the design of the model. It is preferred that aggregates implement the repository to fetch and mutate data and associated objects are available through it.
Provide methods to add and remove objects, which will encapsulate the actual insertion or removal of data in the data store. Provide methods that select objects based on some criteria and return fully instantiated objects or collections of objects whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology. Provide repositories only for aggregate roots that actually need direct access. Keep the client focused on the model, delegating all object storage and access to the repositories. (Eric Evans, 93.)

3.1.13 From concept to services

Event storming and tactical DDD are the two approaches to going forward with defining the business domain to microservices. Although having discussed it with Event Storming inventor Alberto Brandolini the tactical approach did not sound appealing anymore.

Event storming is a process where domain experts, developers and stakeholders sit together and start modeling the business domain by certain pattern that includes read models, commands, events, aggregates, systems, and policies (See figure 1). There are also some supporting notes for actors, hot spots and more. Modeling is done using post-it notes or tools like Miro that enables using virtual notes. Modeling can be done from start to end or end to start, both having their own benefits. The result can be turned straight into application code. There are also tools that turn the event storming model to application code automatically.

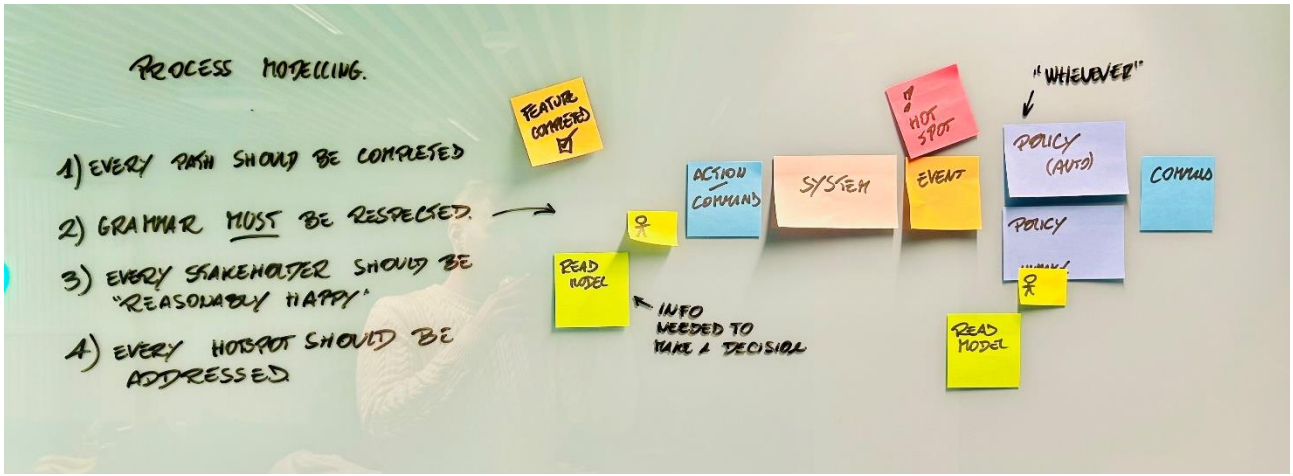


Figure 1. Event Storming

In tactical DDD team starts by analyzing the high-level domain model with domain experts and identifies the business domain and its subdomains. They proceed to define user stories, user scenarios or both for the applicable context. These narratives are later used to identify nouns and system operations. Nouns are answering to what and who and it translates to entities, value objects and services. System operations are actions applicable to the domain. The actor can be human or another service. These operations translate to requests the system handles and come from system requirements.

After the high-level domain model and high-level system operations are analyzed, the team should have a clear view of the domain and its subdomain. Next step is to start adapting the tactical patterns of model driven design. *The tactical patterns are applied within a single bounded context. In a microservices architecture, we are particularly interested in the entity and aggregate patterns. Applying these patterns will help us to identify natural boundaries for the services in our application (AZURE TACTICAL DDD).* The stories and scenarios are broken down into nouns and they are gathered to aggregate and service boundaries. Entities and value objects are placed inside a corresponding aggregate boundary. This work is best done in collaboration with architects and applicable individuals, so thoughts are peer revived during the process. During the process, the verbs are extracted to commands and queries fulfilling the functional requirement defined in the stories. *The behavior of each system operation is described in terms of its effect on one or more domain objects and relationships between them. A system operation can create, update, or delete objects, as well as create or destroy relationships between them (Chris Richardson, 46).* Each command is correlated to aggregate or service. They will be the instructions directly driven through the corresponding service's application programming interface. These commands trigger the state changes for the persistence of object and emit event to message queues.

As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context (AZURE TACTICAL DDD).

3.1.14 Maintaining data consistency across services

Each microservice needing to maintain object state has its own persistent storage solution. The solution can differentiate between services because they are isolated withing the service. Object database solutions are well suited for preserving aggregate state because it is easy to dump the object including related value objects and entities to the storage for later use. Microservice architecture introduces a challenge for data integrity in databases because it is scattered in multiple databases. Database access is capsuled behind system operations provided by service interfaces and distributed transactions are not supported by some database solutions. Distributed transactions are also synchronous and require all services to be available during the transactions. Microservices must maintain data consistency even when availability of services is compromised. With multiple databases the transaction boundary loses isolation and data integrity is impossible to maintain without countermeasures. *Instead of an ACID transaction, an operation that spans*

services must use what's known as a saga, an message-driven sequence of local transactions, to maintain data consistency (Chris Richardson, 111). Saga pattern is an architecture concept for event-driven messaging used in microservice architecture. There are two strategies to implement sagas choreography pattern and orchestration pattern. If a transaction on relational database faces an error during transaction boundary, every transaction made can be rolled back. Sagas contain compensating transactions to either delete the data modification made or to replace it with latest information that is used to a knowledge the failed status of the message. *In many ways, this is another form of what is called eventual consistency. Rather than using a transactional boundary to ensure that the system is in a consistent state when the transaction completes, instead we accept that the system will get itself into a consistent state at some point in the future. (Sam Newman, 91.)*

3.1.15 Saga patterns

Sagas are sequence of events that are triggered by system operations. Saga events perform local transactions between services to maintain data consistency and send events to message broker after each action is performed to notify other services to act accordingly.

Choreography based sagas are usually a better choice because they increase loose coupling. Using choreography-based sagas the service sending the message does not need to know which services are subscribing to its messages. *When using choreography there's no central coordinator telling the saga participants what to do. Instead, the saga participants subscribe to each other's events and respond accordingly. (Chris Richardson, 118.)* Events in choreography-based sagas use asynchronous messaging (See figure 2).

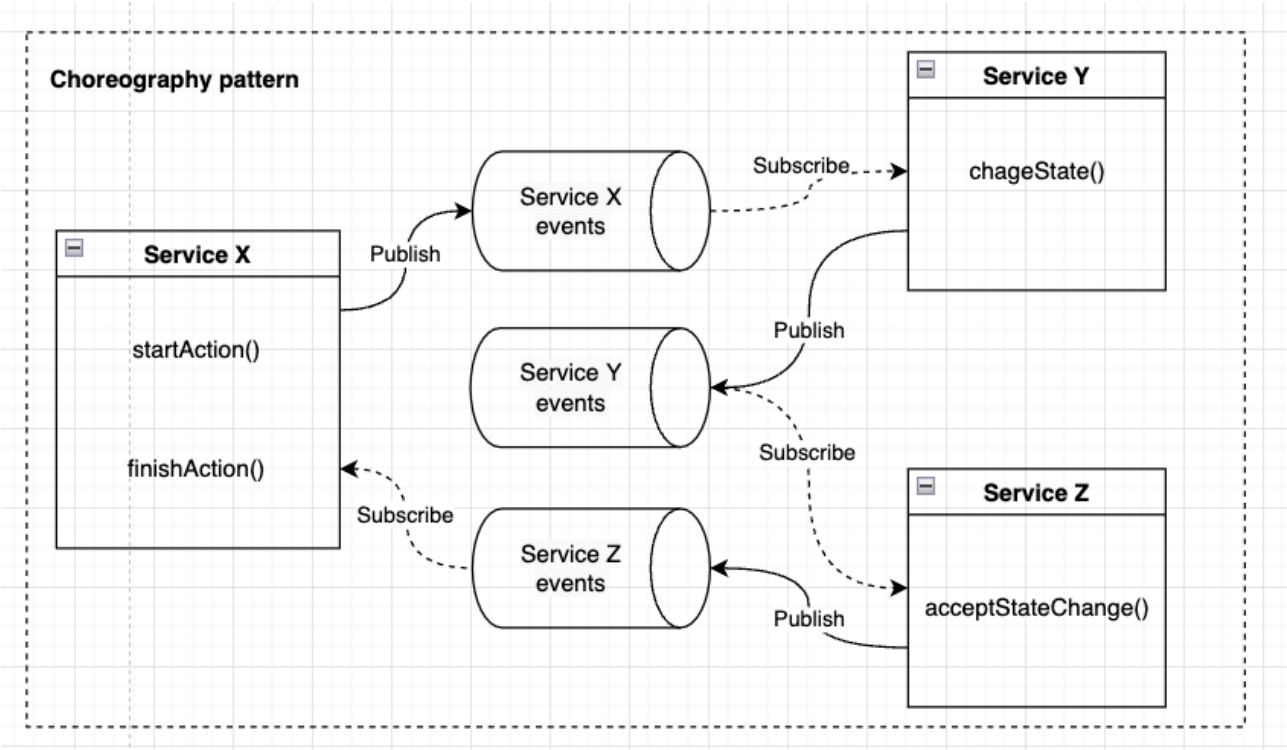


Figure 2. Choreography saga pattern using UML

Orchestration based sagas are more comprehensible to developers because the orchestrator service is doing work synchronously (See figure 3). Doing work synchronously does not necessarily in this context mean that orchestrator service is actively waiting for response because response might take long time to arrive. The orchestrator service can work with both asynchronous and synchronous messages. *Distribute a saga's coordination logic in a saga orchestration class. A saga orchestrator sends command messages to saga participants telling them which operations to perform (Chris Richardson, 119).* Orchestrator performs actions to execute events on remote service and waits for its response before it proceeds to next action. The downside of this pattern is that it needs both the publisher and subscriber services to know about each other. This leads to more coupling leading developers needing to update multiple services after a requirement changes.

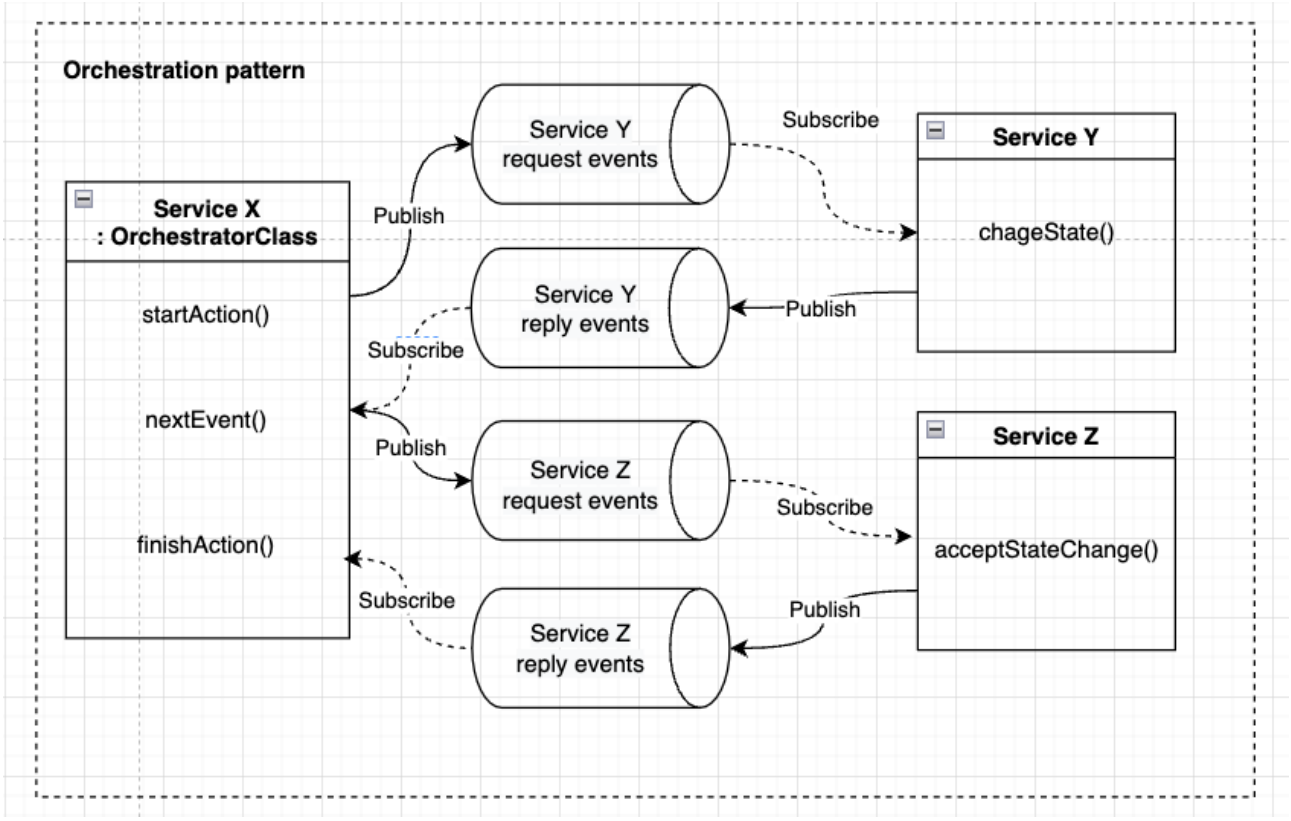


Figure 3. Ochestration saga pattern using UML

3.1.16 Service architecture

Domain driven design allows architects to design architecture using four layers instead of common three-tier architecture. The layers of these architecture patterns are described in the following lists.

1. Presentation layer
2. Logic layer
3. Data layer

1. User interface layer
2. Application layer
3. Domain layer
4. Infrastructure layer

The main difference between three- and four-tier architecture is the isolation of domain layer. Eric Evans describes application layer as *This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in*

the next layer down. (Eric Evans, 41.) Application layer offers interfaces to interact with the service and controls the flow of operations. The domain layer holds the domain objects, their associations and object state which it gets from persistent storage through infrastructure layer.

Hexagonal architecture fits very well to describe and implement four-tier architecture. It introduces concepts such as inbound and outbound adapters. *The Hexagonal Architecture, also referred to as Ports and Adapters, is an architectural pattern that allows input by users or external systems to arrive into the Application at a Port via an Adapter, and allows output to be sent out from the Application through a Port to an Adapter (Pablo Martinez).* Inbound adapters are interfaces to application, and they invoke system operations in the application layer. Outbound adapters perform actions to infrastructure layer and other services. Ports describe the interface of the adapters so other services know how to interact with the service. Hexagonal architecture visualization has driving side which illustrates inbound adapters on left and driven side which illustrates outbound adapters on right (See figure 4). Ports and adapters make it easy to provide multiple user interfaces and infrastructure interfaces to a service simultaneously. For example, the three-tier architecture usually provides only one user interface, and the business logic is built around technologies instead of adapters making it very demanding to change technology choices later.

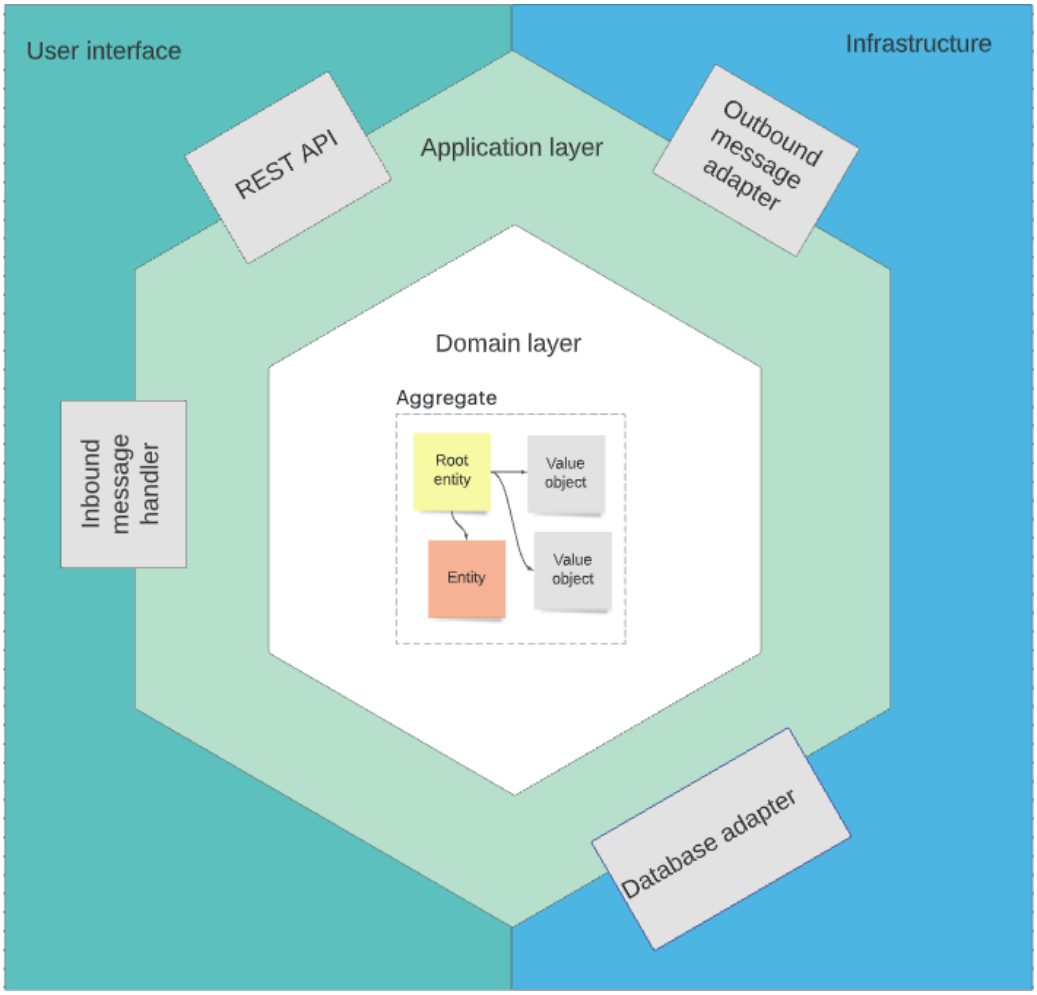


Figure 4. Four-tier architecture in hexagonal architecture pattern

3.1.17 Testing

Microservices increase complexity compared to monolithic systems because the system is distributed across multiple independent services. Automated testing tools are mandatory to identify bugs in the early stages of development. *Continuous Integration (CI) is the process of automating the build and testing of code every time a team member commits changes to version control. CI encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion (Azure what is CI).*

Unit tests are important because they operate at a low level and are fast to run. Unit tests have small scope. They are attached to functions and methods of domain objects and developers must knowledge this when they implement functionality in code. One strategy to ensure that code is

testable is to use test driven development where tests are written before the actual implementation.

Service tests ensure that a service can play its role in a distributed system. Service tests are run after unit tests in CI-pipe because they are faster than end-to-end tests which are run last in CI-pipe. Test automation deploys a service at a time and deploys fake services that collaborate with it. The fake services respond to predefined requests to events that service creates. Results from these tests help the team to understand that messaging is working between services.

End-to-end testing is the most time-consuming part of testing. Implementing and running these tests are slow and difficult and running them requires every part of system to be deployed and available. Failed tests are hard to interpret and often require manual testing to find the cause. There is a positive aspect in running these tests. End-to-end tests the whole scope of system architecture at once including user interface, application interfaces, messaging infrastructure and business logic.

The tests for a given microservice should live in source control with the microservice's source code too, to ensure we always know what tests should be run against a given service (Sam Newman, 107). Having the test in same repository grants the ownership of test to team that owns the service. It is also beneficial to have the test in same place as business logic to benefit from tools that run some of test during file saves or commits to a repository.

3.1.18 Deployment

There are a few suitable options to deploy an application built using microservice architecture. Application can consist of several to hundreds of services that are developed with different programming languages and infrastructure. Continuous delivery and continuous deployment help developers to deploy software rapidly without dependency on another team. DevOps team decides practices and sets up tools to automate deployment when using continuous deployment or let developer to decide when to release continuous delivery pipelines prebuilt and tested software.

Deploying service as virtual machine is a widespread method to deploy software and cloud providers have tools to use this method. In this method the team sets up CD-pipeline to build an image containing the operating system, libraries, and software. This image is then deployed to production. The downside of building an image of whole operating system bundled with the application is that building and deploying is slow and images take a lot of storage space.

Deploying as a docker container is a more recent method to deploy microservices. Docker is an isolated environment containing everything to run application on any host machine. Docker differs from virtual machine by means that containers share the resources with the host machine and other containers. Docker lets developers write infrastructure as code to a docker file and commit it to same repository with the application. This helps to test, develop, and automate the production build of the software. Docker offers a docker compose tool to write and deploy a group of containers within one configuration file. This configuration can contain the application, database, and the message channels. Docker images are deployed to production using CD-pipeline and docker image repository. Docker image repository is a place to publish images which have the base image and the user defined layers, instructions in docker file. After the new image is published the CD-pipeline instructs server to replace the existing containers with the new containers from new image.

3.1.19 API composition and API gateway

API composition pattern is a concept where a microservice works as a composer for data that would normally require multiple queries to backend (See figure 5). *As Chris Richardson (2019, 225) describes API composition. Implement a query that retrieves data from several services by querying*

each service via its API and combining the results.

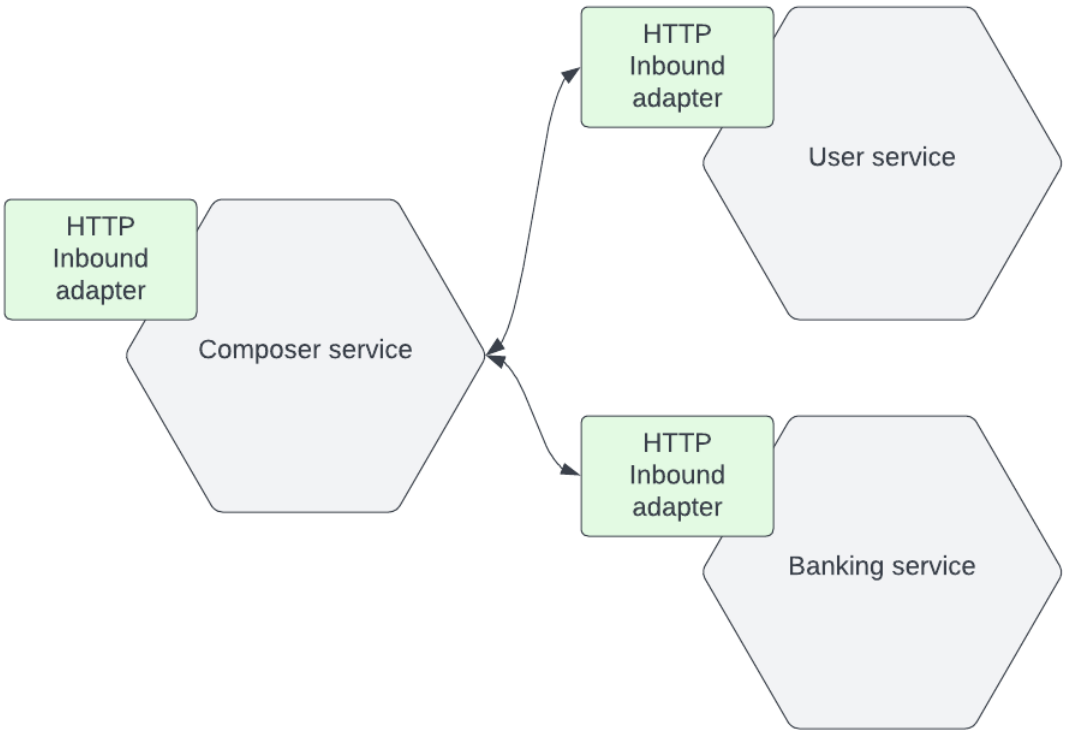


Figure 5. API composition pattern

API gateway functions as API composer but it can do much more (See figure 6). An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting (Use API gateways in microservices). API gateway is an entry point to applications backend, so it is a natural place to perform various tasks. It can be deployed as a microservice, but cloud service providers often provide API composition services to abstract the implementation

behind some configuration. API gateway can also function only as an entry point but leave API composition to a separate composer service like Apollo Router for GraphQL queries (See figure 7).

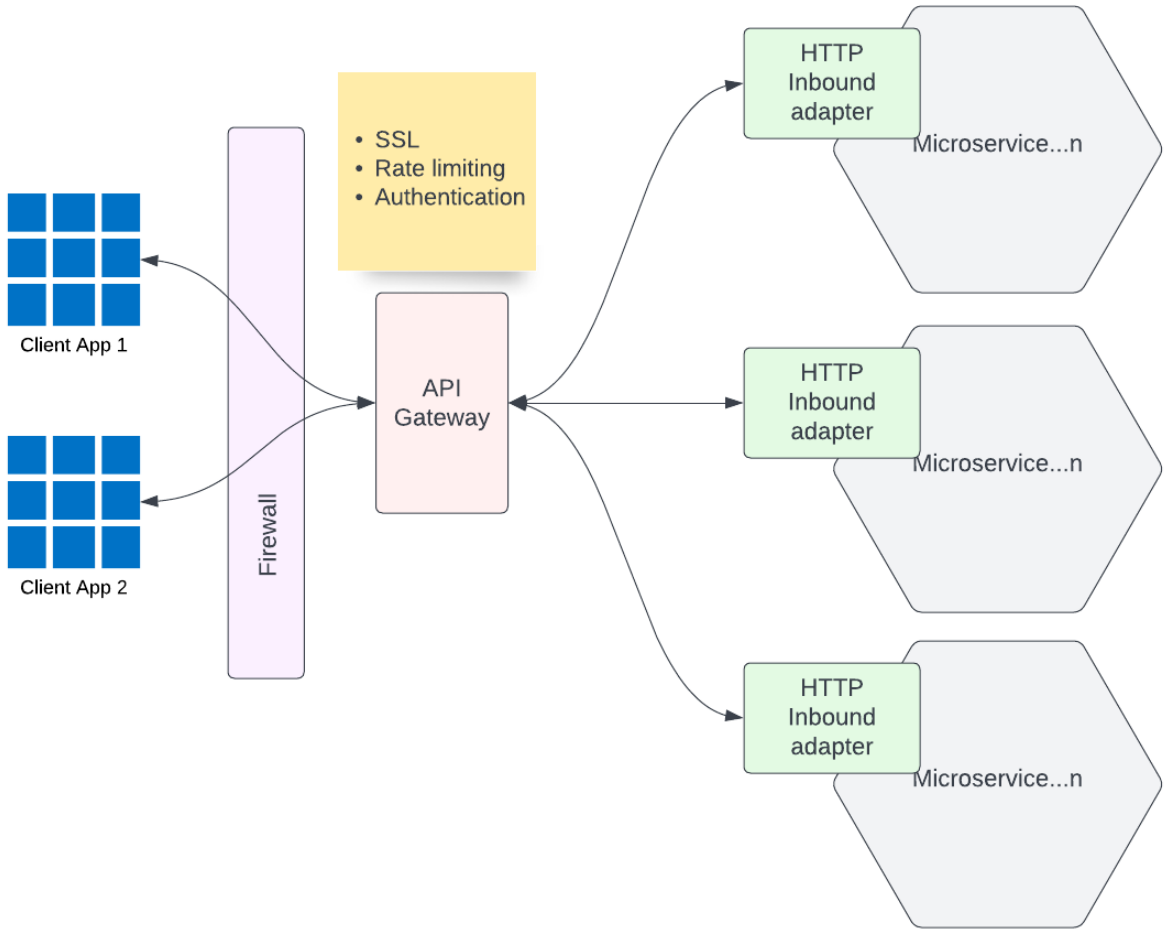


Figure 6. API gateway pattern

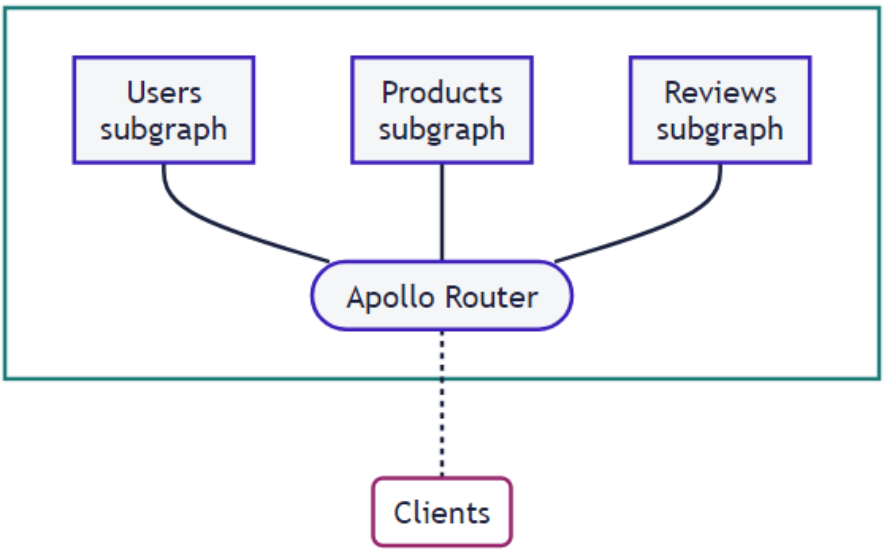


Figure 7. Apollo Router concept

3.1.20 Reliability and resiliency

There are a few key factors to increase reliability of microservice architecture. Handling idempotency and keeping in mind that everything can and will fail helps to build resilient systems. These apply to both synchronous and asynchronous, eventually consistent systems regardless of the way services are deployed.

According to Sam Newman (2011, 215.) The outcome after the first operation is the same even when operations are applied repeatedly. Idempotent operation can be applied multiple times without adverse impact. This is a common and particularly useful way to replay messages in a queue after recovering from error. Idempotent operations are useful both in message and event-based communication and even in traditional REST-API's. Making operations idempotent makes server handle the operation only once resulting to consistency in application data. Making operations idempotent means adding enough information to the operation to distinguish it from other operations.

Preparing for failure on every level is important to keep the system functional in unexpected events. But it is even more important to prepare for the unexpected and that can be achieved by using bulkheads. *Michael T. Nygard describes (2007, 44.) the bulkheads as followingly. In a ship*

there are watertight metal partitions that can be sealed to divide the ship into partitions. The bulkheads prevent water from flowing from section to another when the hatches are closed. This prevents the ship from sinking even when some part of the hull gets penetrated. This same principle can be applied to software by running multiple instances of services simultaneously. When one service fails the rest keep serving the clients. There are tools like Kubernetes that do this automatically and spans new service instances when some of them fail to run. To be even safer the instances can be run on servers in separate locations or even separate continents.

Circuit breakers are a fundamental pattern to prevent systems from failing. The name derives from electric engineering whereas portrayed *by Michael T. Nygard (2007, 43) the circuit breaker protects overloaded electric circuit to not cause the house to get caught in fire. Circuit breaker allows one system to fail without causing the entire system to destroy.* This principle brought to software systems means that in case of some service is unable to handle events, messages or request it starts logging errors and after certain threshold it activates the circuit breaker causing all calls to service to fail until the problem is fixed. After the problem is fixed and the circuit breaker is turned back off all queued calls can be played again to the service.

3.2 Serverless architecture

3.2.1 Introduction

Serverless architecture is an abstraction over traditional microservice architecture. Serverless architecture abstracts the infrastructure and deployment configuration away and that frees developers time to focus on writing code. Every aspect of microservice design principles works equally with serverless microservices. Biggest differences come from deployment, code structure, testing and scalability. Serverless architecture increases availability of application by creating new instance of service for each request and offers better resilience by providing on demand scaling of services.

3.2.2 Architecture concepts

While traditional microservice declares the whole service interface from a single file the serverless architecture enables a different perspective to communicate with the application. A serverless application is built from multiple functions each representing a passage to a service. These

passages are called triggers. A trigger is a function that reacts to one more binding that it is defined to communicate with. Binding is either input or output type and can be used as an inbound or an outbound adapter. Trigger can have an input binding to a HTTP-method to act as an endpoint in REST-API or binding can represent a subscription to events in message broker enabling reactive programming using events. The binding to service or database events makes it easy to use sagas to maintain data integrity, recording events for event sourcing database or for example creating in memory read models for event sourced databases for making efficient queries to a data source. Bindings remove the need for polling the message broker in publish subscribe platforms such as Azure Event Grid.

3.2.3 Drawbacks

Serverless architecture is not a silver bullet to solve all scalability and performance facing architecture problems. Serverless is best suited to asynchronous background processes and tasks that are short lived and not very computing intensive. Serverless can be used to host API-services but developers might have to solve cold start issues when some endpoint are not called frequently enough. Cold start is a state where the serverless function have not been called for long enough time and the service provider have removed the function from reserved slot. At this state the next call to this function leads to runs through warm up phase containing multiple steps to make the code runnable (See figure 8). This process usually takes seconds to finish. *Broadly speaking, cold start is a term used to describe the phenomenon that applications which haven't been used take longer to start up. In the context of Azure Functions, latency is the total time a user must wait for their function. (*Colby Tresness*)*

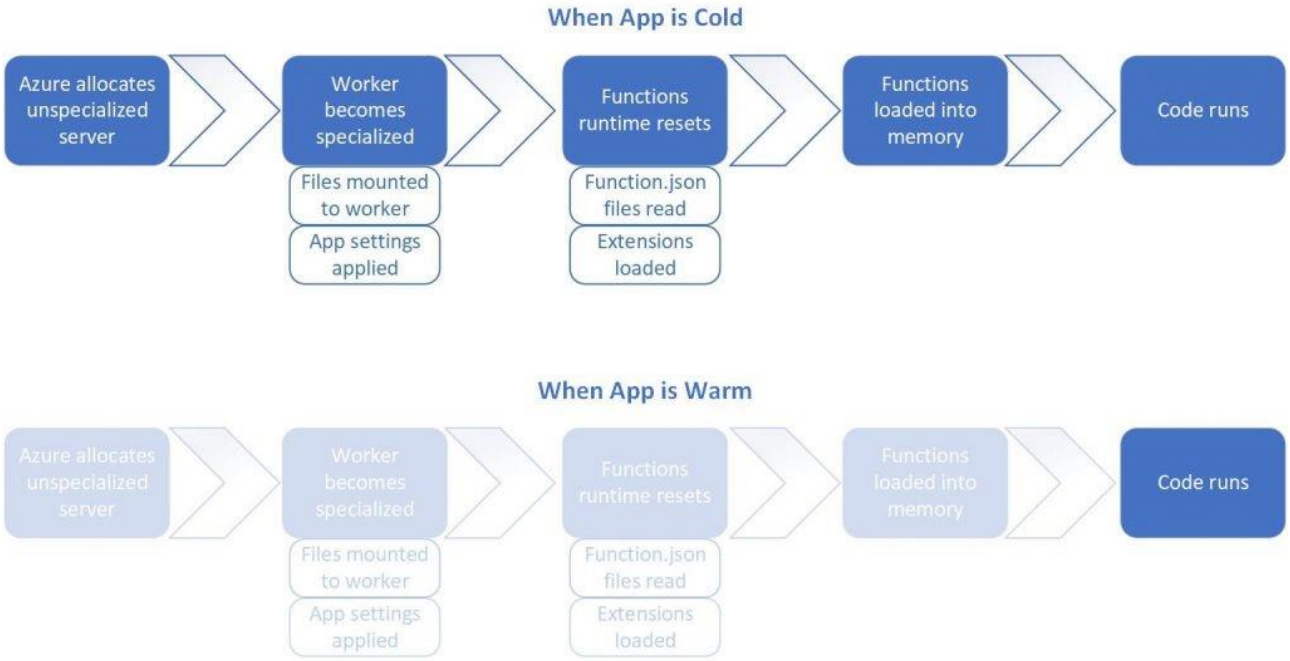


Figure 8. Serverless cold start

Cloud platforms offer paid features to prevent cold starts. For example, Azure offers a premium plan to prevent cold starts and more performance. Paid plans might make sense if resources must

have high availability or for example private networking or private endpoints are required (See figure 9).

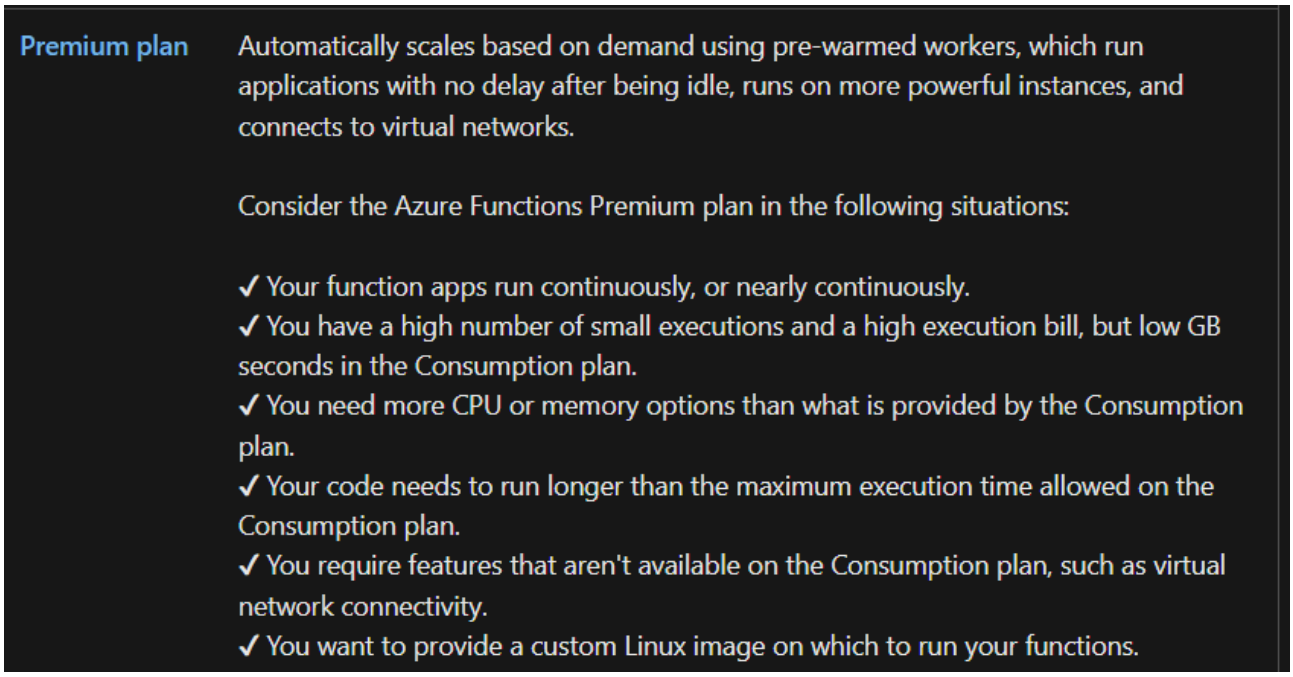


Figure 9. Azure Premium Plan

3.2.4 Deployment

Deployment for serverless functions can be managed by infrastructure as code definition that spans all required functions, databases, and queues for the application. There is infrastructure as code tools like Terraform and Serverless Framework which require a configuration file to be added to code repository. This configuration is like a blueprint for the service being deployed and CI/CD tool then read this file and functions accordingly.

3.3 Monolithic architecture

3.3.1 Introduction

In monolithic architecture there is one codebase and often one deployed instance of service. Codebases are large and multilayered. A large code base is difficult to keep coherent and there is

usually a lot of technical debt especially in old software. Code changes are often difficult to make and test leading to expensive maintenance. Monolithic service is not reliable and resilient. A bug in code or error in server provider can take out the whole service at once. Monolithic software is deployed at once and it only scales poorly. Scaling is done by increasing performance of the hardware so more work can be made in less time or by duplicating the instance and using load balancer to divide request between instances. Scaling can also be made by running identical copies of software but dividing similar things to certain instances. This is common in scaling databases and is commonly known as sharding. Monolithic software usually comes with a monolithic database and has all the data since the software was launched, making the size sometimes hundreds of gigabytes. Database solutions can be scaled using load balancers, read or write replicas or by splitting the database by selected algorithm to multiple databases.

The scale cube (See figure 10) is an effective way to illustrate three-dimensional scaling that microservice architecture solves and what monolithic architecture cannot. Three-dimensional scaling allows applications to scale indefinitely (The Scale Cube). While monolithic architecture scales on X and Z axis the microservice architecture scales on X, Z and Y axis. Y-axis allows each microservice to scale independently. This is a huge benefit as we need to scale only those functionalities of the system that face high traffic instead of buying more performant servers or duplicating the whole application repeatedly.

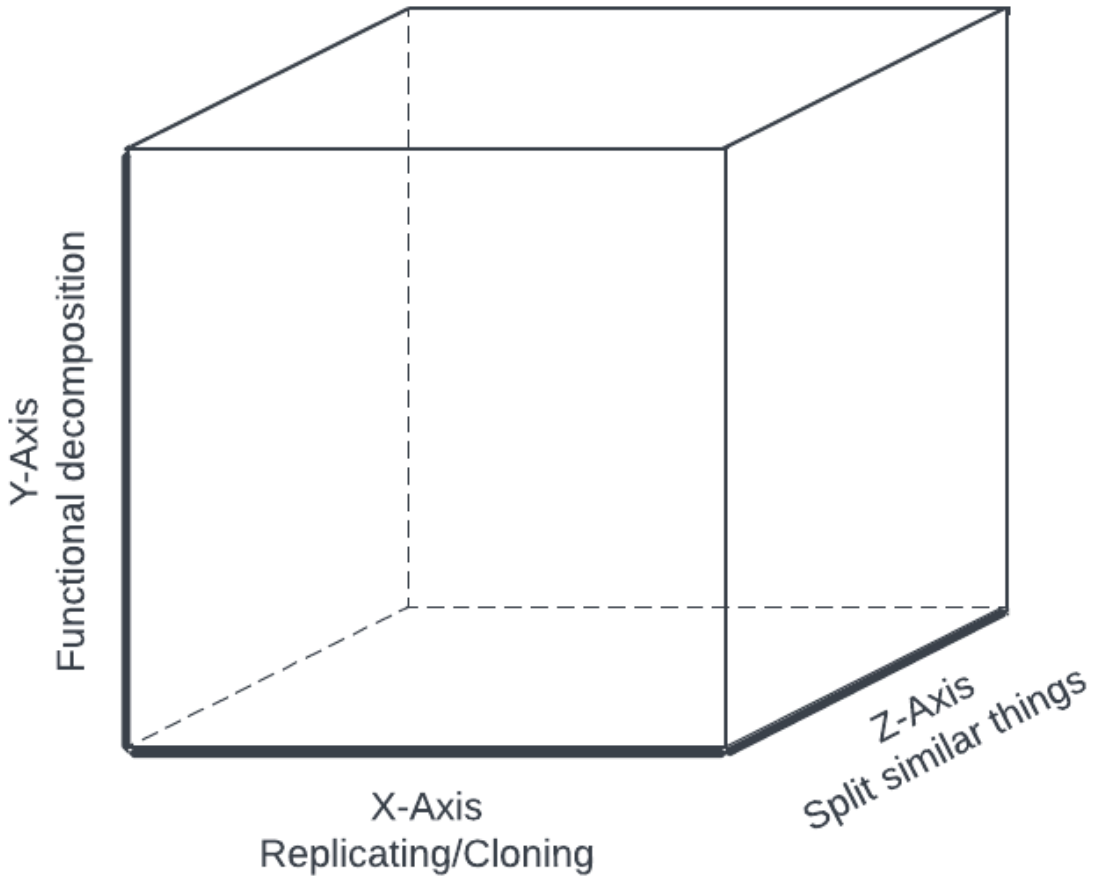


Figure 10. The scale cube

4 Research methods

Research uses triangulation that combines data from multiple research methods of the same study. Qualitative and quantitative data is collected from a survey. The survey is focused on experiences of microservice architecture and is targeted to developers who have field experience from microservices. Action research method is used to capture own experiences and the current state of research of the microservice architecture.

4.1 Data gathering methods

The survey contained ten questions for collecting quantitative data and six questions to collect qualitative data totaling sixteen questions. The qualitative and quantitative questions purposely had a certain scope of overlapping to get some comparative results for the questions. Survey was published to my linked in profile 16.11.2022 and it got answered ten times in next eighteen days with average completion time of ten minutes and thirty-five seconds.

Qualitative questions and their options were as follows:

1. How many years of expertise do you have from software industry?
    a. 0-4
    b. 5-10
    c. More than 10
2. In how many projects you have been using microservice architecture?
    a. 0-4
    b. 5-10
    c. More than 10
3. Which technologies have you used to build and deploy microservices on?
    a. Containers
    b. Kubernetes
    c. Serverless functions
    d. Other
4. Which cloud vendors you have built microservices on?
    a. Microsoft Azure
    b. AWS – Amazon Web Services
    c. GCP – Google Cloud Platform
    d. Alibaba Cloud
    e. Oracle Cloud
    f. IBM Cloud
    g. Digital Ocean
    h. Other
5. How easy have you found the microservices to be worked with?
    a. Numeric input from one to five
6. How performant would describe the microservices you have worked with?
    a. Numeric input from one to five
7. How maintainable the system consisting of micro services have been?
    a. Numeric input from one to five
8. How reliable have you found a system consisting of microservices?
    a. Numeric input from one to five

9. How fast have you experienced the development speed with microservices?
    a. Numeric input from one to five
10. How likely are you to recommend using microservices?
    a. Numeric input from one to five

And qualitative questions were the following:

1. How have the microservice architecture impacted the development speed?
2. How have the microservice architecture impacted the reliability of the systems?
3. How have the microservice architecture impacted the performance of the systems?
4. How have the microservice architecture impacted the maintainability of the systems?
5. What have been the biggest struggle for you in microservice architecture?
6. Which microservice technology would you recommend or invest your time in?

Action research data gathering was conducted by reading experiences from thesis Microservice architecture suitability for contemporary software development by Tomy Salminen and Migrating from Monolithic Application to Microservices from Thi Tran.

5 Results

Respondents' expertise divided followingly. 20% had 0-4 years of expertise, 50% 5-10 years of expertise and 30% had more than ten years of expertise. Every respondent had used microservice architecture in less than five projects. Containers were the trendiest way of deploying micro services in this target group and serverless functions came second (See figure 11).

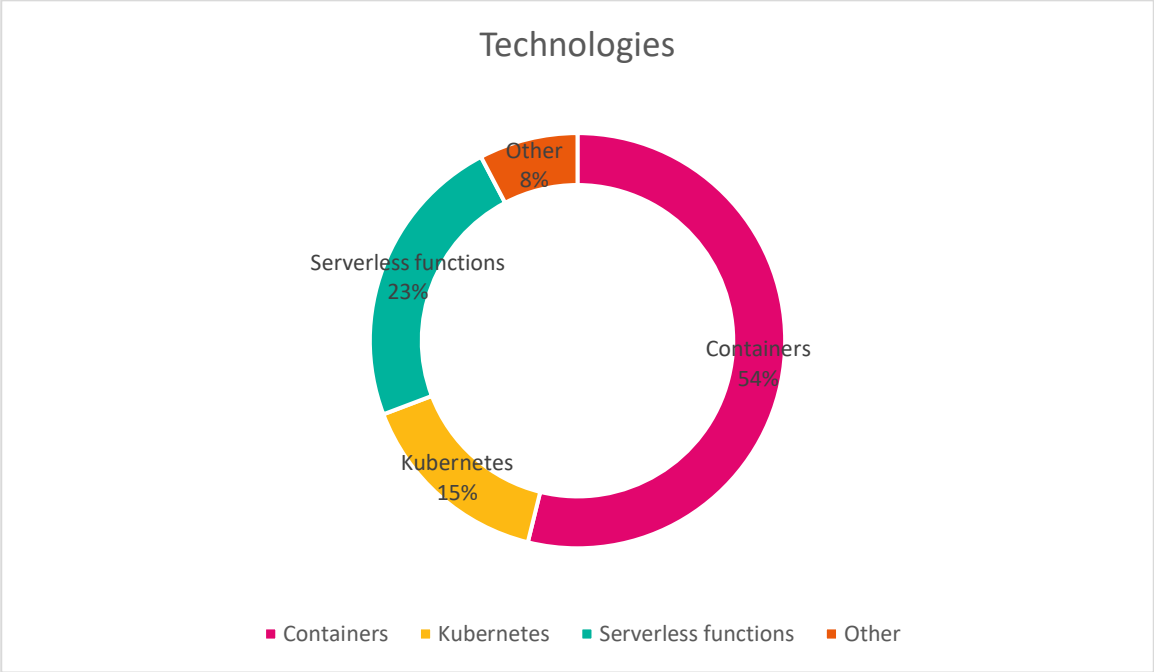


Figure 11. Used Technologies

Microsoft Azure was a leading platform with 60% of the respondents having experience of using it and AWS came close second with 40% and Digital Ocean with 20% experiences within target group (See chart 12). There was one vote for the other in this category. This could potentially be something like Microsoft Orleans, Akka Toolkit or Kalix by Lightbend which are frameworks for building message driven applications. It would have been interesting to include question of these to questionnaire.

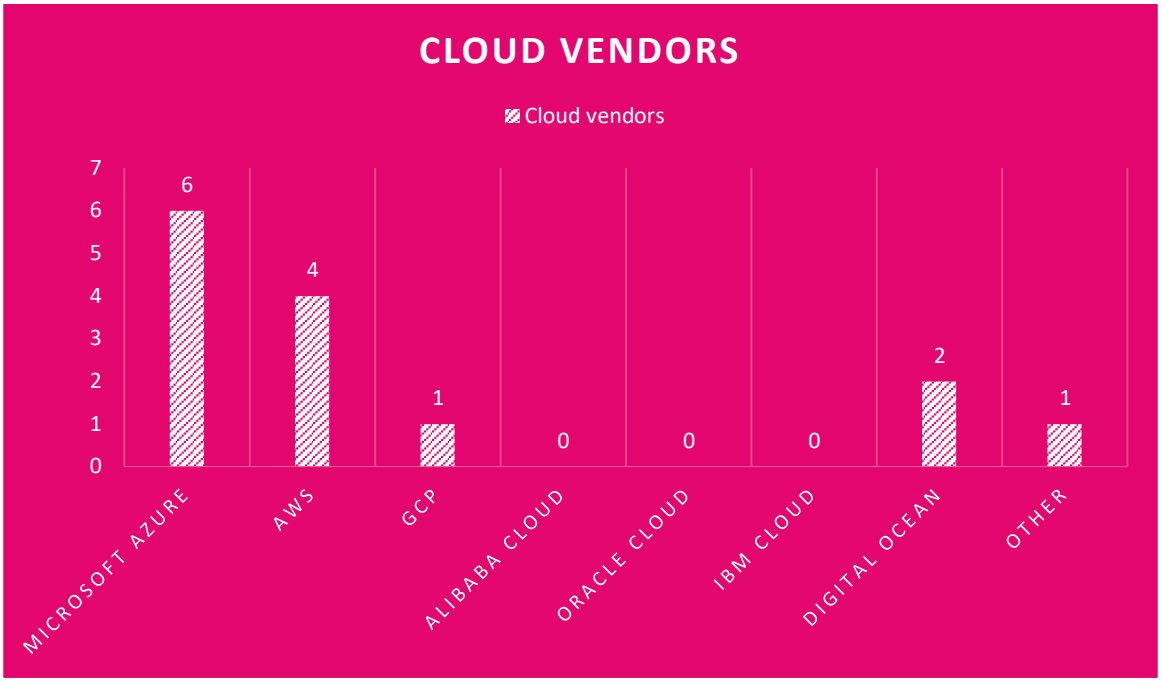


Figure 12. Cloud vendors

Microservice architecture introduces an elevated level of complexity and knowledge of cloud vendors to build working cloud infrastructure for the application and this is visible in the survey responses as most respondents answered 3/5 for how easy they have found microservices to be worked with and responses averaging 3.1/5.

Performance and scalability are one of the key factors when switching to microservice architecture and microservices seem to reclaim this promise. Quantitative data from question about how performant the microservices have been averaged to 4/5 and having worst score of 3/5. Qualitative data showed that respondents experienced systems to perform like other systems they had been working with. Most answers pointed out that they had not experienced noticeable differences.

Maintainability of the systems built with microservice architecture in this target group averaged to 3.5/5 but having large spread in responses. Analysis of qualitative data raised the controversiality of the subject. Some respondents felt that microservices had only positively impacted maintainability and then others pointed out that microservices had either increased difficulty to

maintain systems or made it equally difficult as maintaining monolithic system because of added complexity of monitoring services.

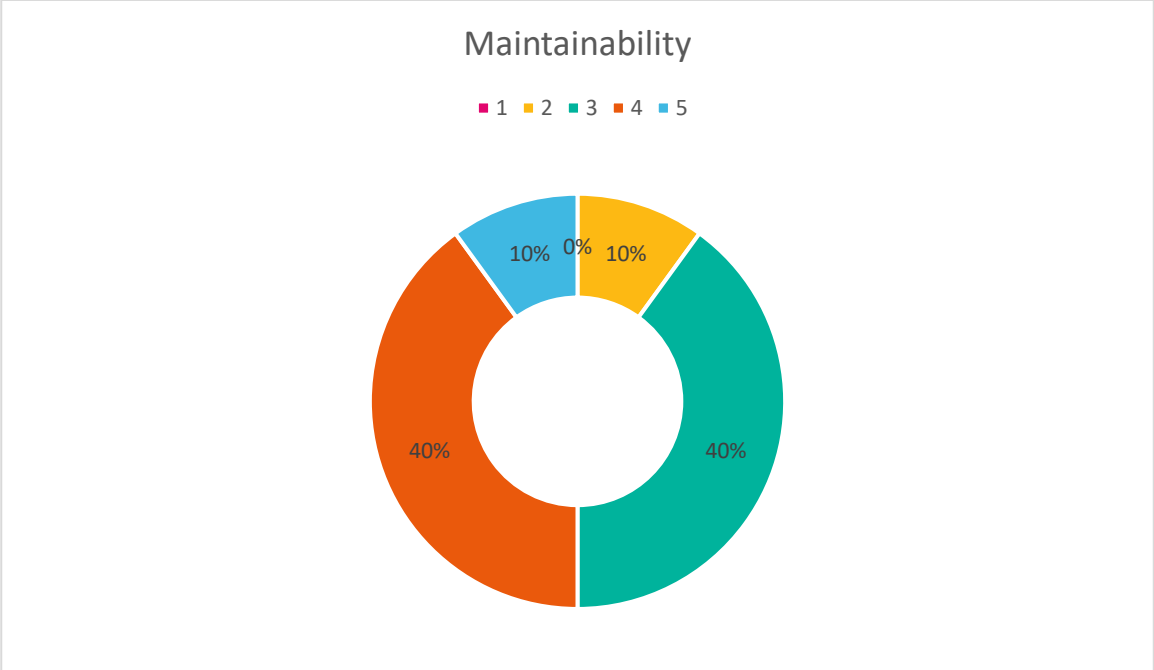


Figure 13. Maintainability of a system consisting of microservices

Respondent found that microservices are quite reliable with an average score of 3.8/5. Qualitative data suggested that if microservices are independent and their dependencies are professionally managed the reliability increases because if one service fails it will not affect the entire system.

Development speed is one of the issues with monolithic applications. In crucial application there might be code written by people who have left the company, and no one wants to touch that code. Respondents scored the development speed from 2 to 5 averaging 3.5/5. It is impossible to say if the bottle neck has been complexity, communication, or something else.
Qualitative data for development speed pointed out mostly negative impact to development speed. Reasons being high amount of event-based communication making testing harder, poor architectural design, high learning curve and then a couple of answer pointing out that after a certain period the development speed normalized to similar levels as it had been before microservice architecture. One respondent had experienced that development speed went down

but the quality increased due to better maintainability and less complexity. This complexity could derive from monolithic codebase becoming a big ball of mud in the long run.

One of the most interesting parts of the survey was qualitative data from the question of what the biggest struggle in microservice architecture has been. Responses varied from technology, documentation, and communication issues. Some experienced that defining the service boundaries and contracts between services is hard and other respondents pointed out that start to think along domain driven principles and figuring out what each service must do has been difficult. These responses show that that turning the domain knowledge to code is very demanding. The asynchronous nature of microservice architecture and poor isolation of services had caused trouble for some of the respondents.

Respondents were noticeably confident to recommend using microservice architecture averaging 4.25/5. This aligned with the other results gives a strong signal that microservice architecture is a promising path for modern software development. The last question addressed which technology the respondents would recommend investing your time in. Containers and serverless were the most frequently used words in the result set. Most responses were cloud platform-oriented showing that platforms like AWS and Azure are a direction where developers should invest their time in.

Tomy Salminen did a case study on a Company named Futurice Oy. The target was to use microservices to exchange data between two systems. Data needed heavy modification, so the services worked as an anticorruption layer between two bounded contexts. Microservices were running in Node environment and were deployed to containers in Amazon web services. AWS S3 was used as temporary data storage and events were delivered using AWS Simple Queue Service.

According to Tomy Salminen the developing environment was challenging due to having multinational teams, challenging communication patterns and rapid change of requirements. Even though the challenging environment the developing cycle was fast, and they managed to keep the communication overhead down. Having continuous integration with test automation helped with the complexity of microservice architecture.

The system was built with scalability in mind, but it was not needed as the data source could not scale with the consuming services. However, the performance of the system was fast enough to process typical data set within minutes so there was no need to start working with this issue as that performance was more than enough for the task.

Research from Thi Tran examined the transition from monolithic architecture to microservice architecture by decomposing it to microservices by establishing a strangler pattern – a pattern to keep the old system functioning while bringing the new services on the side. The monolithic system was an e-commerce application running on Node runtime and deployed on GCP. The process was monitored to by tracking the uptime of the system and conducted by containerizing the monolith and deploying it to GPC. The availability or the whole systems was 97,4% and a bit better because there were some errors in the tracking process. The developing experience was good according to Thi Tran. He started from easiest bounded context and moved to more complex ones later. He faced problems with setting up environments and service migration and migrating data to distributed databases.

5.1 Analysis

The results of the survey point out an incredibly positive impact to developing experience and the quality of the applications. Qualitative and quantitative data collected from the survey align with the case study by Tomy Salminen and Thi Tran. Both the survey and Tomys case study noted the problems with added complexity and communication. Qualitative data raised issues with poor architectural design, tight coupling, asynchronous nature of event driven systems and difficulties applying domain driven design to work in their domain. This's work proved that a monolithic system can be decomposed with a good availability using a stranger pattern.

Survey respondents were confident to recommend using microservice architecture as most of them had positive experiences of helping with the maintainability, performance, and development speed of the systems they had been working with. A case study by Tomy Salminen suggests similar as they tackled complex communication patterns and delivered code fast to production with excellent quality events, though the developing environment was challenging. Thi Tran suggests that strangler pattern is recommended way for transition from monolithic architecture to microservice architecture.

Futurice Oy is using AWS platform for delivering their solution to customer. This follows the pattern shown in the survey results as most of the respondents had used either Azure or AWS in their work and they were also the most recommended platform in the survey. The e-commerce platform in Thi Tran's research was deployed on Google Cloud Project platform that was the fourth most used vendor in the survey.

6 Conclusion

6.1 Reliability of the research

The survey was conducted on the Linked In social media platform. Linked In is a network where I can reach my work colleagues and their associates. The survey did not get a lot of responses but then on the other hand, most of the respondents answered each of the questions and the data did not contain any responses that seemed fake.

Statistically the data is not remarkable as there were only ten people who completed the survey. Results are aligned with the Thi Trans's research, a case study by Tomy Salminen and my own findings from working with distributed event-based systems for multiple years. The qualitative and quantitative data collected in a survey is credible and can be used as directional information when used with other data sets.

6.2 Ethical analysis

This research have followed the commonly approved code of ethics (Arene Ry) by collecting survey results anonymously and by handling the data appropriately by the initial plan. Respondents were informed that the survey data is collected toward use in this thesis. Publishing these results do not cause risk or harm to any organization or individuals. Results of the qualitative and quantitative data are displayed the way that they protect the respondents anonymity.

Claims in the theoretical part are backed up by referencing public work in this area of research and using credible sources by well-known authors if available.

6.3 Examination of key results in relation to the initial theoretical framework

Theoretical framework states that microservice architecture when done right increases software quality by multiple factors. The survey was focused on the few main topics: performance, maintainability, developing experience and reliability. The case study by Tomy Salminen cover only some of these topics. Thi's research covered mostly reliability and developing experience.

Developing experience should increase by removing communication overhead as teams can work independently in their own bounded context. This appears to be the case in both the case study by Tomy Salminen and the survey I conducted in linked In. Also Thi Tran suggests that developing experience in decomposing monolith to microservices was fairly good.

Maintainability of the system should increase as the systems are built from multiple independent modules. Survey respondents had both neutral and positive experiences from maintainability of microservices. Survey respondents suffered from poor architectural design, bad error handling, and added complexity of the microservice architecture.

Performance of microservices should withstand high amount of traffic because they allow multidimensional scaling. Even though a single request can be slower than in a monolithic application the response times should stay low even in high demand. This was proven true in both the Tomy's case study and the survey. Responses in the survey stated that most respondents had not experienced noticeable difference in response times of systems built on microservice architecture.

6.4 Suggestions for improvement

The extent of survey be quite a lot bigger to get more credibility to the results. These should be more questions like which advanced platforms respondents have been using. Because it is more efficient to use frameworks which abstract away CQRS, event sourcing, queues and other architectural patterns used in microservice architecture. These frameworks let developers focus on business logic instead of the infrastructure and architecture.

There is room for finding more research of developing experience and performance of distributed systems and case studies of monoliths that have been decomposed to microservices. See how the results compare with the survey results and Tomy Salminen's case study.

One way to get another point of view to results is to build a benchmark software that executes similar functionality on serverless functions, containerized microservices, serverless containers and a monolithic system. Then benchmark these systems with small load and high load and see how they perform. How the systems affect response times, how many clients can be server simultaneously and what happens if some function goes to non-responding state.

References

Opinnäytetyön eettiset ohjeet. (n. d). Arene Ry. https:/www.arene.fi/wp-content/uploads/Raportit/2018/arene-opinnaytetyoprosessin-eettiset-suositukset_muistilistat-opiskelijalle-ja-ohjaajalle.pdf

Azure Functions hosting options. (2022, November 23). Microsoft Learn. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale

Bhandari, P. (2022, January 3). Triangulation in research. https://www.scribbr.com/methodology/triangulation

Evans, E. (2001). Domain Driven Design tackling complexity at the heart of a software. Addison-Wesley Professional.

Khaitan, N. (2022, November 15). Medium Blog.Why Use Microservices — Breaking the Monolith. https://medium.com/towards-polyglot-architecture/why-use-microservices-breaking-the-monolith-ce90d47430cb

Martinez, P. (2021, July 7). Hexagonal architecture, there are always two sides to the story. Medium Blog. https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c

McGlothin, R. (2018, April 25). The Scale Cube. Akf Partners Blog. https://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/

Newman, S. (2014). Building Microservices Designing fine-grained systems. O'Reilly Media, Inc.

Nygard, M.T. (2007). Release It! Design and Deploy Production Ready Software. Pragmatic Bookshelf.

Richardson, C. (2019). Microservice patterns. Manning Publications Co.

Salminen, T. (2018). Microservice architecture suitability for contemporary software development. https://urn.fi/URN:NBN:fi:amk-2018053111438

The Apollo Router. (n. d). Apollo GraphQL Docs. Retrieved December 3, 2022 from https://www.apollographql.com/docs/router

Tran, T. (2020). Migrating from Monolithic Application to Microservices. https://urn.fi/URN:NBN:fi:amk-2020052714294

Tresness, C. (2018, July 2). Understanding serverless cold start. Azure Blog. https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start

Use API gateways in microservices. (n. d). Microsoft Learn. Retrieved December 2, 2022 from https://learn.microsoft.com/en-us/azure/architecture/microservices/design/gateway

Using tactical DDD to design microservices. (2022, March 3). Microsoft Docs. https://docs.microsoft.com/en-us/azure/architecture/microservices/model/tactical-ddd

What is Continuous Integration? (2021, May 15). Microsoft Docs. https://docs.microsoft.com/en-us/devops/develop/what-is-continuous-integration

Appendices

Appendice 1. Aggregation of survey results

Experiences of microservices in work life

10	10:35	Active
Responses	Average time to complete	Status

1. How many years of expertise do you have from software industry?

0-4	2
5-10	3
More than 10	5

2. In how many project you have been using microservice architecture?

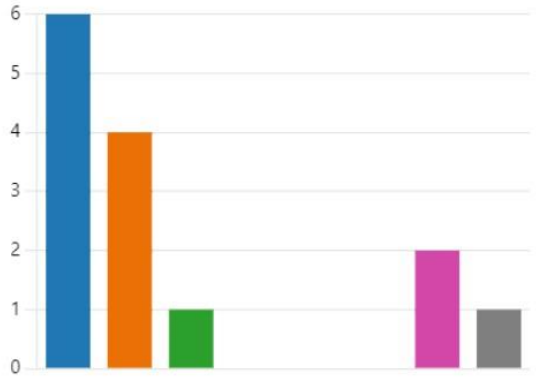
0-4	10
5-10	0
More than 10	0

3. Which technologies have you used to build and deploy microservices on?

Containers	7
Kubernetes	2
Serverless functions	4
Other	1

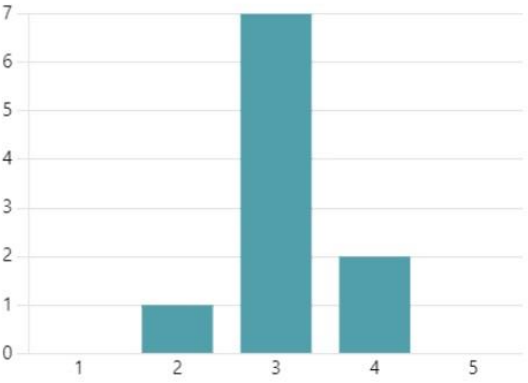
4. Which cloud vendors you have built microservices on?

Microsoft Azure	6
AWS - Amazon Web Services	4
GCP - Google Cloud Platform	1
Alibaba Cloud	0
Oracle Cloud	0
IBM Cloud	0
Digital Ocean	2
Other	1



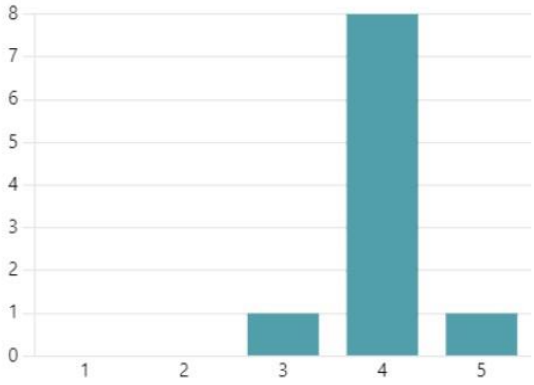
5. How easy have you found the microservices to be worked with?

3.10
Average Rating



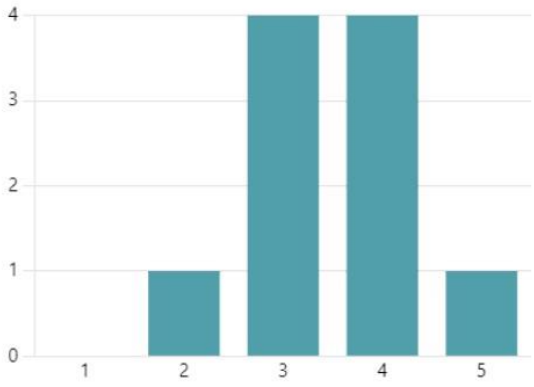
6. How performant would describe the microservices you have worked with?

4.00
Average Rating



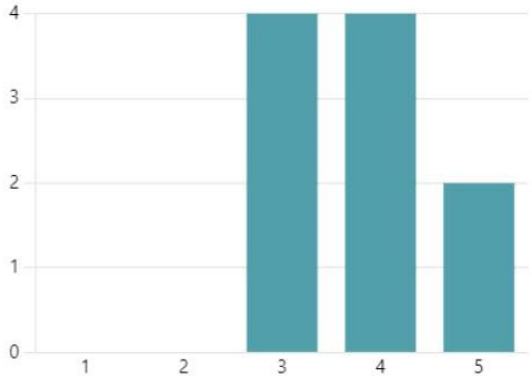
7. How maintainable the system consisting of micro services have been?

3.50
Average Rating



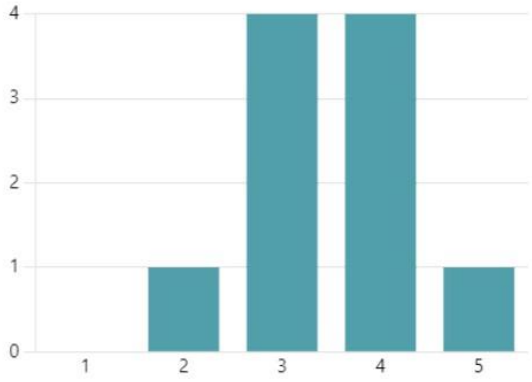
8. How reliable have you found a system consisting of microservices?

3.80
Average Rating



9. How fast have you experienced the development speed with microservices?

3.50
Average Rating



10. How have the microservice architecture impacted the development speed?

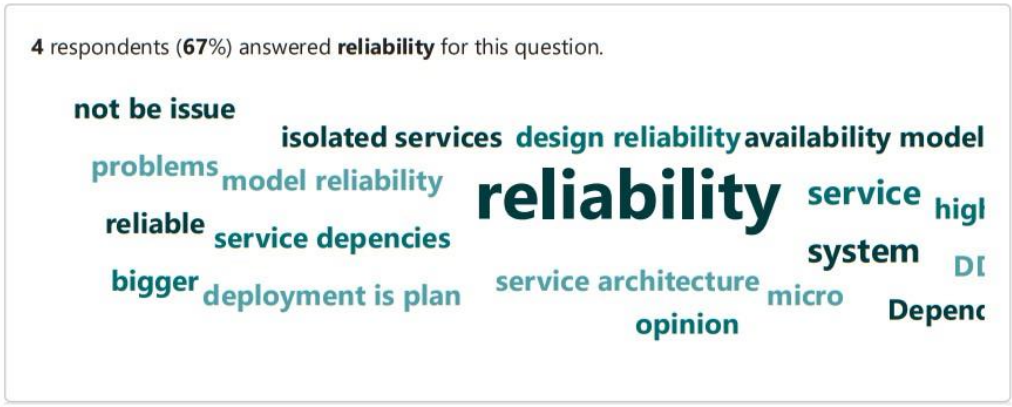
7
Responses

Latest Responses
"It has been a little slower, but in the end when you can use t..."

11. How have the microservice architecture impacted the reliability of the systems?

6
Responses

Latest Responses
"No change so far."



12. How have the microservice architecture impacted the performance of the systems?

6
Responses

Latest Responses

"No change but perhaps it will be easier to scale."

4 respondents (**67**%) answered **performance** for this question.

13. How have the microservice architecture impacted the maintainability of the systems?

6
Responses

Latest Responses

"It has made it more complex, architect is required. With arch...

4 respondents (**67**%) answered **system** for this question.

14. What have been the biggest struggle for you in microservice architecture?

6
Responses

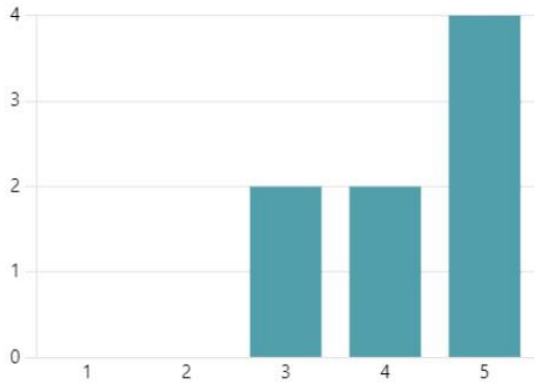
Latest Responses

"When a service changes and gets a new revision the change...

4 respondents (**67**%) answered **services** for this question.

15. How likely are you to recommend using microservices?

4.25
Average Rating



16. Which microservice technology would you recommend or invest your time in?

5
Responses

Latest Responses