Final Year Project

Performance Analysis of Design Patterns in Microservice Architecture

Rajit Banerjee

Student ID: 18202817

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science with Data Science

Supervisor: Professor John Murphy

UCD School of Computer Science

University College Dublin

29th November 2021

Contents

1 Project Specification 2

1.1 Problem Statement 2

1.2 Background 3

1.3 Related Work 3

1.4 Datasets 3

1.5 Resources Required 3

2 Related Work and Ideas 4

2.1 Microservices 4

2.2 Software design patterns 5

2.3 Common design patterns in microservice architecture 5

2.4 Issues and challenges with microservices 8

2.5 Performance engineering 8

2.6 Summary 9

3 Project Work Plan 10

Chapter 1: Project Specification

1.1 Problem Statement

Microservice architecture is a style of designing software systems to be highly maintainable, scalable, loosely-coupled and independently deployable. Moreover, each service is built to be self-contained and implement a single business capability. Design patterns in software engineering refer to any general, repeatable or reusable solution to recurring problems faced during the software design process. The aim of this project is to analyse the performance of a number of microservice design patterns (based on metrics such as query response time, CPU/RAM usage, cost of hosting and packet loss rate), and evaluate their benefits and shortcomings depending on the business requirement and use case. A non-exhaustive list of design patterns that could be explored is as follows:

- API Gateway
- Chain of Responsibility
- Asynchronous Messaging
- Database or Shared Data
- Event Sourcing
- Command Query Responsibility Segregation (CQRS)
- Saga
- Circuit Breaker
- Strangler (Decomposition)
- Consumer-Driver Contract Test
- Externalise Configuration
- Aggregator
- Branch

For the aforementioned design patterns, sufficiently complex simulations will be designed for the performance engineering experiments. The project will also look at some common issues in microservices, and how they compare with traditional monolithic architectures.

1.2 Background

Microservices have gained traction in recent years with the rise of Agile software development and a DevOps [1] approach. As software engineers migrate from monoliths to microservices, it is important to make appropriate choices for system design and avoid "anti-patterns". Although no one design pattern can be called the "best", the performance of systems can be optimised by following design patterns suited to the use case, with the right configuration of hardware resources.

1.3 Related Work

Due to their popularity, microservices have been written about extensively in books like [2], [3], [4]. Articles such as [5], [6], [7], [8], [9] discuss the intricacies of microservice architecture as well as the trade-offs between various common design patterns. In [10], the performance problems inherent to microservices are explored, with evaluations performed using a custom-built prototyping suite. Akbulut and Perros [11] dive into the performance analysis aspect of microservices that is being proposed in this project, where they consider 3 different design patterns.

1.4 Datasets

Any data that is to be used or analysed in this project will be generated during the course of experiments. There are no dependencies on additional datasets.

1.5 Resources Required

A non-exhaustive list of resources is specified below, following preliminary needs assessment.

- Languages/Frameworks: Node.js + Express.js, React.js
- Tools: Git, Docker, Apache JMeter
- Database: MongoDB
- Compute: Linux server maintained by the UCD School of Computer Science

Chapter 2: Related Work and Ideas

The aim of this chapter is to provide the readers with a holistic view of microservices, design patterns and performance evaluation, especially highlighting some of the important terminology and latest developments in the field. The discussion will consider the existing literature which illustrate how the topics have been previously explored, including the significance of performance engineering for microservices.

2.1 Microservices

Divide and conquer, the idea of breaking down complex systems into smaller manageable parts, is an ancient and proven paradigm which has been applied to computer science since the early 1960s. To tackle the complexities of software systems, concepts such as *modularity* and *information hiding* were introduced in 1972 by D. Parnas [12], as well as *separation of concerns* by E.W. Dijkstra in 1974 [13].

Regarding the term "microservices", the two most acknowledged definitions were given by Lewis and Fowler [9] and Newman [4], both around the same time in 2014. Newman considers microservices as a particular way of implementing SOA (Service-oriented Architecture) well, whereas Lewis and Fowler define it as a new architectural style contrasted against SOA:

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

The aforementioned definitions are first compared in 2016 by Zimmermann [14], who contrasted them with existing SOA principles and patterns. Since then, a number of systematic mapping studies have been published ([15], [16], [17]) to identify existing literature on microservices, in an effort to bridge the gap between academia (lack of many significant publications) and industry (racing past academia in terms of popularising and adopting microservices). In particular, Di Francesco et al. [17] cite both the prior studies conducted by Pahl and Jamshidi [15] and Alshuqayran et al. [16], and mention that due to the *bottom-up* approach (practical solutions first) that the industry has taken with microservices, many "fundamental principles and claimed benefits have still to be proven". By learning from practical applications, the industry has identified best practices, however aspects such as performance, functional suitability and maintainability of microservices at the industry-scale (instead of small-scale examples) are yet to proven in academia.

In the past decade or so, the adoption of microservices has been accelerated by the success of companies such as Amazon Web Services (AWS) [1], Netflix [2], Spotify [3] and Uber [4]. Amazon also has a whitepaper describing how microservices can be implemented using AWS cloud services, taking into consideration ways to reduce operational complexity and design distributed systems components (service discovery, data management, configuration management, asynchronous communication, and monitoring) [18].

[1] https://aws.amazon.com
[2] https://www.netflix.com
[3] https://www.spotify.com
[4] https://www.uber.com/

2.2 Software design patterns

In software engineering and related fields, *design patterns* are generally defined as reusable solutions to commonly occurring problems in software design. Although design patterns cannot be directly converted to code (like an algorithm described in pseudo-code), they provide a blueprint on how a problem can be approached in various situations. Unlike *algorithms*, design patterns are not meant to define any clear set of instructions to reach a target, but instead provide a high level description of an approach. The characteristic features and final result are laid out, however the actual implementation of the pattern is left up to the requirements of the business problem and use case. Every "useful" design pattern should describe the following aspects: the intent and motivation, the proposed solution, the appropriate scenarios where the solution is applicable, known consequences and possible unknowns, as well as some examples and implementation suggestions.

Over half a decade of software engineering experience has taught developers that it is indeed rare to come across a hurdle that hasn't been crossed before in some shape or form. Most obstacles and day-to-day decisions would have been tackled previously by another developer, thanks to which the idea of *best practices* has been formed over the years. Such solutions are accepted as superior, as they save time, are adequately efficient, and don't have many unknown side effects.

The most widely known literature on the topic is the 1994 textbook [19] by the Gamma, Helm, Johnson and Vlissides (Gang of Four), which is considered as the milestone work that initiated the concept of software design patterns. The authors, inspired by Christopher Alexander's definition of patterns in urban design [20], describe 23 classic patterns that fall under 3 main categories: *creational, structural*, and *behavioral* patterns.

- Creational patterns such as *Factory Method, Builder* and *Singleton* increase the flexibility and reusability of code by providing object creation mechanisms.
- Structural patterns such as *Adapter, Bridge* and *Facade* illustrate the process of building large, flexible and efficient code structures.
- Behavioral patterns such as *Chain of Responsibility, Iterator*, and *Observer* provide guidelines to distribute responsibilities between objects, and are specific to algorithms.

In recent times, design patterns have had a tendency of coming across as somewhat controversial, primarily due to a lack of understanding about their purpose. In this regard, it is important for developers to note that in the end, design patterns are merely guidelines and not hard-and-fast rules that must be conformed to.

2.3 Common design patterns in microservice architecture

Several attempts have been made to apply the concepts of software design patterns to microservices and categorise commonly seen patterns. In Fig. 2.1, C. Richardson provides a number of pattern groups, including *Decomposition, Data management, Transactional messaging, Testing, Deployment, Cross-cutting concerns, Communication style, External API, Service discovery, Reliability, Security, Observability* and *UI* [21].

Similarly, M. Udantha describes 5 different classes of design patterns applicable to microservices, namely *Decomposition, Integration, Database, Observability* and *Cross-cutting concerns* (see Fig. 2.2).

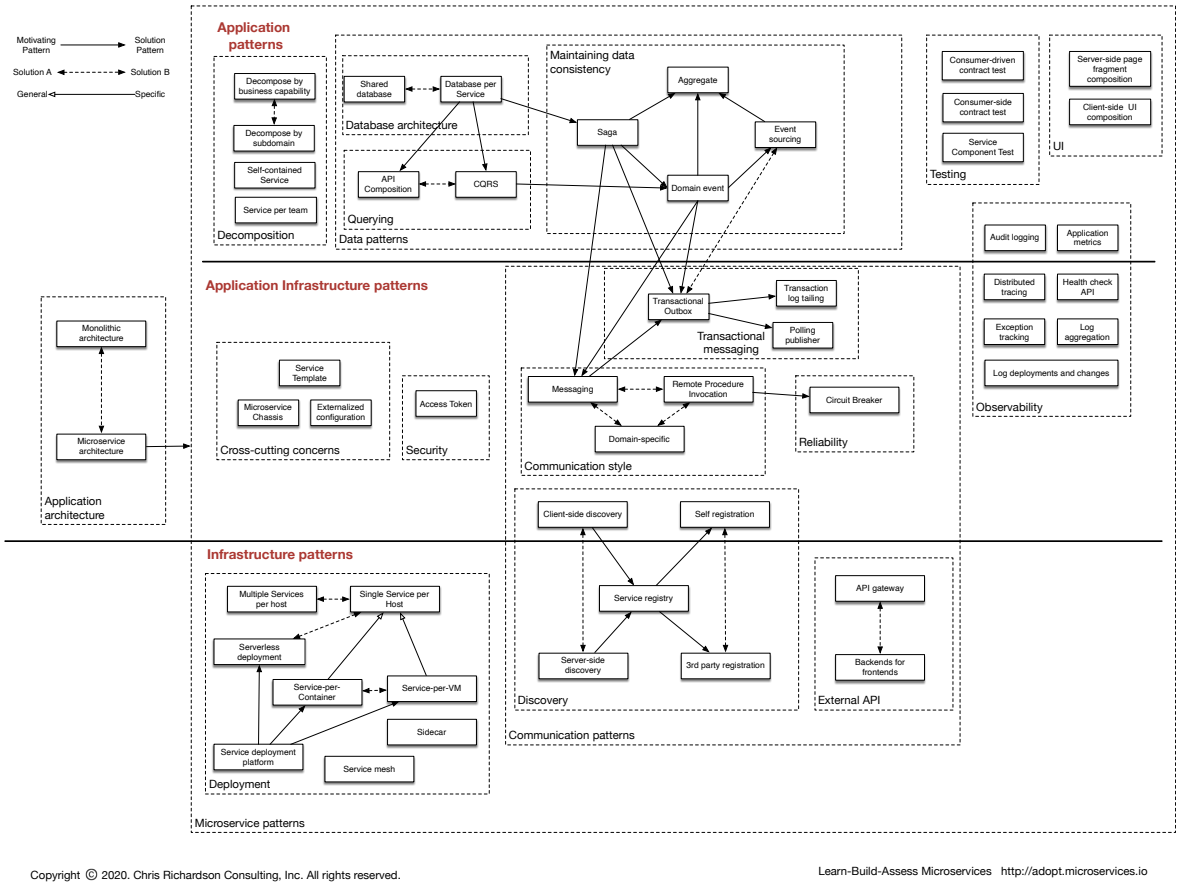

Figure 2.1: Groups of microservice design patterns [21].

It is important to note that despite minor differences in the categorisation of patterns, the groups suggested by the authors above are generally along the same lines, and aim to address the common principles of microservice design, such as scalability, availability, resiliency, flexibility, independence/autonomy, decentralised governance, failure isolation, auto-provisioning and CI/CD (continuous integration and delivery) [8].

- *Decomposition* patterns lie at the heart of microservice design, and illustrate how an application can be broken down by business capability, subdomain, transactions, developer teams, etc. They also include refactoring patterns that guide the transition from monoliths to microservices.
- *Data management* patterns guide the design of database architecture (e.g. whether multiple services will share a database or each service will get a private database). They also lay out methods for maintaining data consistency, dealing with data updates and implementing queries.
- *Integration* patterns include API gateways, chain of responsibility, other communication mechanisms (e.g. asynchronous messaging, domain-specific protocols), as well as user interface (UI) patterns.
- *Cross-cutting concern* patterns describe ways of dealing with concerns that cannot be made completely independent, and result in a certain level of tangling (dependencies) and scattering (code duplication). Examples include externalising configuration, handling service discovery (client-side such as Netflix Eureka [5]; server-side like AWS ELB [6]), using circuit

[5] https://github.com/Netflix/eureka
[6] https://aws.amazon.com/elasticloadbalancing/

Figure 2.2: Udantha's 5 classes of microservice design patterns [8].

breakers, or practising Blue-Green deployment (keeping only one of two identical production environment live at any time). Service discovery and circuit breaker (for reliability) are also considered as communication patterns.

- *Deployment* patterns illustrate multiple ways of deploying microservices, including considerations about hosts, virtual machines, containerisation, number of service instances, as well as serverless options.
- *Observability* is a part of performance engineering, and such patterns are essential to any form of software design, since application behaviour must be continuously monitored and tested to ensure smooth working. Aggregating logs, keeping track of performance metrics, using distributed tracing, and maintaining a health check API are some invaluable practices which aid the troubleshooting process.

It is interesting to note that there are certain similarities between the *GoF*'s creational, structural and behavioral patterns [19] and the aforementioned microservice-specific patterns.

Apart from the sources mentioned above, there have been some studies conducted to recognise architectural patterns for microservice-based systems. In [22], Bogner et al. perform a qualitative analysis of SOA (Service-oriented Architecture) patterns in the context of microservices. Out of 118 SOA patterns (sources: [23], [24], [25]), the authors found that 63% were fully applicable, 25% were partially applicable and 12% were not all applicable to microservices. Taibi et al. [26] tackle the issue of inadequate understanding regarding the adoption of microservice architectures. The authors explore a number of widely adopted design patterns, under the categories of *Orchestration and Coordination, Deployment* and *Data storage*, by elaborating the advantages, disadvantages and lessons learnt from a number of case studies. Thus, a catalogue of patterns is presented, all constituents of which demonstrate the common structural properties of microservices as discussed earlier.

2.4 Issues and challenges with microservices

Although microservice architectures have a number of benefits, it is important to note that they are not universally applicable to solve all problems at scale. Moreover, there are various trade-offs to consider, and *anti-patterns* ("bad practices") to avoid when building new microservice-based systems or transitioning from monolithic applications.

In [10], K. Cully investigates whether unforeseen performance issues can be introduced in a healthy system by independent communication and resiliency configuration of microservices. Using a custom-built rapid prototyping suite, Cully shows that multiple operational constraints in microservices can be conflicting, and optimising the performance within one part of microservice-based system can lead to major pitfalls in other parts.

M. Fowler argues in [27] that transitioning from a well-defined monolith to an ecosystem of microservices has various operational consequences and complexities, which demand certain competencies such as rapid resource provisioning (characteristic of cloud-native applications), observability and monitoring setup, CI/CD, as well as DevOps culture in the organisation. Fowler goes on to elaborate on some of the trade-offs that teams face when choosing microservices over monoliths. The costs associated with microservices include dealing with the distributed nature of the system, with issues such as slow remote calls, risk of failure, consistency, as well the increased operational effort required by teams [28].

Finally, it is important to avoid common anti-patterns that are known to be counterproductive when adopting the microservice architectural style. A few examples of such practices include [29], [30]:

- Distributed monolith - a monolith refactored into several smaller services, all of which are interdependent (tightly coupled).
- Dependency disorder - services must be deployed in a particular order to work.
- Shared database - resource contention due to all microservices sharing a single data store.
- API gateway - not using an API gateway, or using it incorrectly by building a gateway in every service.
- Entangled data - all services get complete access to all database objects (not following the principle of least privilege).
- Improper versioning - services such as APIs not designed for changes and upgrades, leading to reduced maintainability.

2.5 Performance engineering

Having discussed the background research and ideas related to microservice-based systems and design patterns, it is now appropriate to narrow down the focus to performance engineering, especially in the context of microservices and their design patterns. Software performance engineering (SPE) is a vital part of the software design lifecycle, including, but not limited to performance simulation and modelling (e.g. using UML [7] diagrams), benchmarking, monitoring, and performance testing.

[7] https://www.uml.org

2.5.1 Distributed systems

Innovation in performance engineering has often been driven by breakthroughs in industry, out of necessity. In fact, Netflix is well known for pioneering the concept of *chaos engineering* as early as 2011, when migrating to the cloud (AWS). To address the inadequacy of available resilience testing, the company invented a suite of tools (known as the "Simian Army" [31]), the most significant of which is Chaos Monkey [8]. Chaos engineering tests the performance of a large-scale system when subjected to experimental failure scenarios, especially in production environments. Similarly, there is the principle of ownership at Amazon, where it is the development team's responsibility to handle the entire software lifecycle including operations, performance testing, and monitoring ("you build it, you run it" [32]).

Much work has been done over the years on studying performance engineering techniques for distributed systems in general.

2.5.2 Evaluation of microservice-based systems

2.6 Summary

- Mention limitations of the related works

[8] https://netflix.github.io/chaosmonkey/

Chapter 3: Project Work Plan

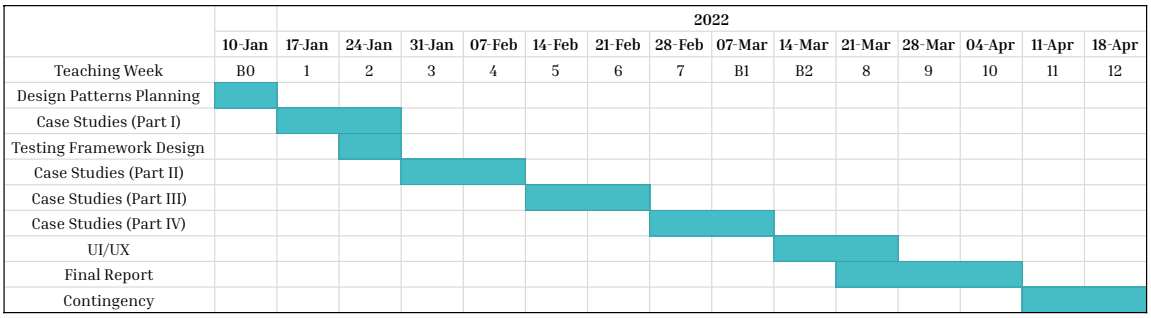

Figure 3.1: Gantt chart for project timeline

- **Design Patterns Planning (1 week)**: The initial planning here is of particular significance and will decide the direction and pace of the project's implementation phase. Various microservice design patterns will be considered to select a few important patterns which can be easily demonstrated using simulations, and then performance tested.
- **Testing Framework Design (1 week)**: Designing a simple testing framework (e.g. load testing plans) during the first case study will facilitate the adoption of similar strategies for subsequent case studies.
- **Case Studies (Parts I, II, III, IV) (8 weeks)**: These case studies will form the bulk of the project, and will be split into four parts for convenience, each comprising a set of related design patterns (each group of case studies will possibly address 2-3 patterns). The majority of effort required here will be concerned with the back-end development of dummy microservice-based systems using containers (Docker). Evaluation, both qualitative and quantitative, is well integrated with the implementation phase of the project, since performance analysis/testing will be carried out in tandem with the case study experiments. Different configurations used during performance testing will facilitate the discussion around suitability and aptness of a number of microservice design patterns.
- **UI/UX (2 weeks)**: A simple web user interface will be designed to visualise the results of performance testing conducted for microservices during the different case studies, and also provide a cost-benefit analysis of the design patterns under investigation.
- **Final Report (3 weeks)**: Although the core parts of the report should be written as progress is made with tasks, a dedicated period is set aside for refinement and completion.
- **Contingency (2 weeks)**: Time set aside to be used only in the event of unforeseen issues or challenges.

Bibliography

[1] Amazon Web Services. "What is DevOps?" [Online]. Available: https://aws.amazon.com/devops/what-is-devops/ (visited on 26th Oct. 2021).

[2] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, Oct. 2018, Book.

[3] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, Mar. 2017, Book.

[4] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Dec. 2014, Book.

[5] M. Kamaruzzaman. "Effective microservices: 10 best practices," [Online]. Available: https://t.co/ZM78yg19OR?amp=1 (visited on 26th Oct. 2021).

[6] M. Kamaruzzaman. "Microservice architecture and its 10 most important design patterns," [Online]. Available: https://towardsdatascience.com/microservice-architecture-and-its-10-most-important-design-patterns-824952d7fa41 (visited on 26th Oct. 2021).

[7] S. Kappagantula. "Everything you need to know about microservices design patterns," [Online]. Available: https://www.edureka.co/blog/microservices-design-patterns (visited on 26th Oct. 2021).

[8] M. Udantha. "Design patterns for microservices." (30th Jul. 2019), [Online]. Available: https://dzone.com/articles/design-patterns-for-microservices-1 (visited on 26th Oct. 2021).

[9] J. Lewis and M. Fowler. "Microservices: A definition of this new architectural term." (25th Mar. 2014), [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 26th Oct. 2021).

[10] K. Cully, "Performance problems inherent to microservices with independent communication and resiliency configuration," M.S. thesis, University College Dublin, Ireland, Mar. 2020.

[11] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019. DOI: 10.1109/MIC.2019.2951094.

[12] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, Dec. 1972. DOI: 10.1145/361598.361623.

[13] E. W. Dijkstra, "On the role of scientific thought," 1974. [Online]. Available: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html.

[14] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 32, Nov. 2016. DOI: 10.1007/s00450-016-0337-0.

[15] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," Jan. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[16] N. Alshuqayran, N. Ali and R. Evans, "A systematic mapping study in microservice architecture," Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[17] P. Di Francesco, P. Lago and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, Apr. 2019. DOI: 10.1016/j.jss.2019.01.001.

[18] Amazon Web Services, "Implementing Microservices on AWS," Tech. Rep., 9th Nov. 2021. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html (visited on 28th Nov. 2021).

[19] E. Gamma, R. Helm, R. Johnson and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994, ISBN: 0201633612.

[20] C. Alexander, S. Ishikawa and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977, ISBN: 0195019199.

[21] C. Richardson. "A pattern language for microservices," [Online]. Available: https://microservices.io/patterns/index.html (visited on 28th Nov. 2021).

[22] J. Bogner, A. Zimmermann and S. Wagner, "Analyzing the Relevance of SOA Patterns for Microservice-Based Systems," Mar. 2018.

[23] T. Erl, *SOA Design Patterns*. Boston, MA, USA: Pearson Education, 2009.

[24] T. Erl, B. Carlyle, C. Pautasso and R. Balasubramanian, *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*, ser. The Prentice Hall service technology series. Prentice Hall, 2012.

[25] A. Rotem-Gal-Oz, *SOA Patterns*. Shelter Island, NY: Manning Pubications Co., 2012.

[26] D. Taibi, V. Lenarduzzi and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study," in *CLOSER*, SciTePress, 2018, pp. 221–232.

[27] M. Fowler. "Microservice Prerequisites." (28th Aug. 2014), [Online]. Available: https://martinfowler.com/bliki/MicroservicePrerequisites.html (visited on 29th Nov. 2021).

[28] M. Fowler. "Microservice Trade-Offs." (1st Jul. 2015), [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html (visited on 29th Nov. 2021).

[29] V. Alagarasan. "Seven Microservices Anti-patterns." (24th Aug. 2015), [Online]. Available: https://www.infoq.com/articles/seven-uservices-antipatterns/ (visited on 29th Nov. 2021).

[30] J. Kanjilal. "Overcoming the Common Microservices Anti-Patterns." (2nd Nov. 2021), [Online]. Available: https://www.developer.com/design/solving-microservices-anti-patterns/ (visited on 29th Nov. 2021).

[31] Netflix Technology Blog. "The Netflix Simian Army." (19th Jul. 2011), [Online]. Available: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 29th Nov. 2021).

[32] C. O'Hanlon, "A Conversation with Werner Vogels: Learning from the Amazon Technology Platform," pp. 14–22, 2006. DOI: 10.1145/1142055.1142065.