Assignment of master's thesis

Title:	Analysis and Comparison of Application Architecture: Monolith, Microservices and Modular Approach
Student:	Bc. Martin Skalický
Supervisor:	doc. Ing. Tomáš Vitvar, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Microservices architecture is widely today used to design and develop applications. However, developers do not always use this architecture correctly which may lead to performance or management issues, or issues with application sustainability. The goal of this work is to compare application design practices using the classic monolithic architecture and microservices architecture and further explore hybrid design approaches.

The work will include the following parts:

- Analysis of monolithic, microservices, and modular architecture design approaches and their comparison along the lines of performance, maintainability, sustainability, and testability.
- Design of a methodology to design and develop applications for new projects.
- Verification and evaluation of the proposed methodology in a sample project (Proof of Concept, PoC) which will include the implementation of a sample application.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 19 September 2023 in Prague.

Master's thesis

ANALYSIS AND COMPARISON OF APPLICATION ARCHITECTURE: MONOLITH, MICROSERVICES AND MODULAR APPROACH

Bc. Martin Skalický

Faculty of Information Technology
Department of Software Engineering
Supervisor: doc. Ing. Tomáš Vitvar, Ph.D.
January 10, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Martin Skalický. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Skalický Martin. *Analysis and Comparison of Application Architecture: Monolith, Microservices and Modular Approach.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **vi**

Declaration **vii**

Abstract **viii**

List of abbreviations **ix**

1 Architectures **3**
1.1 Monolith 4
1.1.1 The Single process Monolith 4
1.1.2 The Modular Monolith 4
1.1.3 The Distributed Monolith 5
1.1.4 Service Oriented Architecture (SOA) 6
1.1.5 Characteristics 7
1.2 Modulith 8
1.2.1 Characteristics 9
1.3 Microservices 10
1.3.1 Brief history 11
1.3.2 Not the proclaimed silver bullet 12
1.3.3 Characteristics 14
1.4 Summary matrix 15

2 Financial well-being project **17**
2.1 Introduction to the project 17
2.2 Infrastructure 18
2.3 Microservices architecture 18
2.4 Characteristics 19
2.4.1 Performance 19
2.4.2 Maintainability 20
2.4.3 Sustainability 20
2.4.4 Testability 20
2.4.5 Complexity 21
2.5 Summary 21

3 Proof of concept **23**
3.1 Application 23
3.2 Monolith example 24
3.2.1 Database 25
3.3 Modulith example 25
3.3.1 Database 26
3.4 Microservices example 26
3.4.1 Database 27
3.5 Benchmark methodology 27

3.6 Benchmark results . . . 29
3.6.1 Performance scenario . . . 29
3.6.2 Latency scenario . . . 30
3.7 Summary . . . 32

4 Methodology **33**
4.1 Why Modulith over Microservices . . . 34
4.2 Monolith to Modulith . . . 34
4.3 Modulith to Hybrid Modulith . . . 36
4.4 Modulith to Microservice . . . 36
4.5 Microservice to Modulith . . . 36
4.6 Summary . . . 37

5 Conclusion **39**

Attached media contents **43**

List of Figures

1.1 A single process monolith: all code is packaged into a single process. [6] 5
1.2 In a modular monolith, the code inside the process is divided into modules. [7] . 5
1.3 Services are delivered into service inventory (right) from which service composition are drawn (bottom). 7
1.4 Mondulith architecture. Modules encapsulate logic and its own data. 9
1.5 A microservice exposing its functionality over a REST API and a topic. [15] . . . 12
1.6 Overview of architectures. *Modifications to API always affects the whole application. 16

2.1 Inter-microservice dependency graph of the backend after about 3 years of development. Each node represents a microservice and each arrow represents a direct dependency on another microservice. 22

3.1 Diagram describes client flow of application. 24
3.2 Database schema displaying tables and relations of Monolithic example. 25
3.3 Architecture diagrams for application examples build with architectures: Monolith, Modulith and Microservices. 28
3.4 Benchmark flow diagram. 29

4.1 Conversion steps from Monolith to Hybrid Modulith 35
4.2 Conversion step from Modulith to Microservices. 37
4.3 Conversion step from Microservices to Modulith. 38

List of Tables

3.1 Table containing benchmark results comparing Monolith, Modulith and Microservices. Microservices and much more CPU reserved, since it consist of 5 services (every singe one has 0.5 CPU assigned). 30
3.2 Table containing benchmark for Modulith and Microservices with module invoice moved into separate service running in multiple number of instances (indicated by number in parentheses). 30
3.3 Table containing benchmark results comparing Monolith, Modulith and Microservices for second scenario. 31
3.4 Table containing benchmark results comparing Monolith, Modulith and Microservices for second scenario. 32

I would like to thank my Supervisor for the valuable time he spent with me and for all he's done to advise me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Praze on January 10, 2024 . . . .

Abstract

Microservices architecture have become a ubiquitous standard in the software industry, where they are often used as a one-size-fits-all solution, disregarding their drawbacks. This thesis aims to analyse the design approaches of monolithic, microservices and modern modular architecture and compare them, while emphasising the frequently overlooked negative consequences of Microservices in favour of Modular Monolith architecture. The author discusses his experience working on microservices projects and the challenges they faced with the architecture in practice. A proof of concept application is developed for each type of architecture mentioned and thoroughly analysed for performance and latency. Finally, the thesis concludes by presenting a methodology for employing the Modular Monolith architecture approach in new projects and how the said architecture can evolve throughout the application lifecycle.

Keywords Modulith, methodology, Microservices, software archtiecture

Abstrakt

Architektura mikroslužeb se stala všudypřítomným standardem v softwarovém průmyslu, kde se často používá jako univerzální řešení, aniž by se braly v úvahu její nevýhody. Cílem této práce je analyzovat přístupy k návrhu monolitické architektury, architektury mikroslužeb a moderní modulární architektury a porovnat je a zároveň zdůraznit často přehlížené negativní důsledky mikroslužeb ve prospěch modulární monolitické architektury. Autor se zabývá svými zkušenostmi s prací na projektech mikroslužeb a problémy, se kterými se v praxi s touto architekturou setkal. Pro každý ze zmíněných typů architektury je vytvořena aplikace, která je důkladně analyzována z hlediska výkonu a latence. V závěru práce je představena metodika pro využití přístupu architektury Modular Monolith v nových projektech a způsob, jakým se může uvedená architektura dále rozvíjet v průběhu životního cyklu aplikace.

Klíčová slova Modulith, metodologie, Mikroslužby, softwarová architektura

List of abbreviations

ESB	Enterprise service bus
PoC	Proof of Concept
SOA	Service Oriented Architecture

Introduction

Software architecture is the foundation of every application. Like foundation is construction, it has profound effect on the quality of what is build on top of it. As such, it holds a great importance in terms of the successful development and maintenance of the whole system. Architecture serves as a blueprint for a system. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components. Since the dawn of software development, the Monolith architecture has been the industry's favored solution. However, the increasing complexity of applications has brought to light its limitations in terms of scalability, modularity and maintenance. Since then, there has been a search for a new type of architecture that will overcome these limitations.

Several approaches have been tried, such as Service Oriented Architecture, which seemed promising, but none have matched the popularity of the Monolith. However, more than ten years ago, the Microservices architecture emerged and has spread extensively across the industry, becoming the new standard. Microservices shine at solving the most pressing problem which has risen with constantly increasing number of application consumers, the scalability by allowing to scale horizontally individual parts of the system on demand. The entire application is split into small components with limited responsibility called "microservice", which further improves maintenance and makes the system easier to understand.

Unfortunately, Microservices constitute a type of distributed system that introduces significant complexity to the application and complicates deployment. They undoubtedly offer great benefits for sizeable projects with many development teams, yet smaller and medium-sized projects frequently overlook alternative options like Modular architecture. It falls midway between Monolith and Microservices, combining the effortless deployment of Monolith and modularity of Microservices without the complexity of distributed system. The scalability can be achieved later as well by utilizing hybrid approaches.

The objective of this thesis is to elucidate the frequently overlooked disadvantages of Microservices and recommend modern modular alternatives. This is imperative as Microservices can significantly increase project expenses, impede development progress and even result in project failure in the worst-case scenario. The first chapter of the thesis is devoted to a detailed analysis of the Monolith, Modular and Microservices architectures, culminating in a comparative analysis in relation to several factors such as communication, maintenance, deployment and complexity. The second chapter presents my personal experience working on a project that utilized Microservices architecture while highlighting positive as well as negative aspects of the architecture. In the third chapter, an example application is created for each type of architecture and comprehensively analyzed concerning performance and latency. The thesis concludes with the creation of a methodology for selecting the appropriate architecture for a new project and how it can be applied and further modified over the course of the application's lifecycle.

Chapter 1
Architectures

This chapter discusses three different software architectures: the classic monolithic architecture, the popular Microservices, and what can be found somewhere in between: Modulith. Architectures will be defined and compared between the following lines:

- **Performance** defines how much workload application can handle, how much latency is present during processing and how well application can scale.
- **Maintainability** defines degree to which application is understood, repaired and enhanced from technological perspective. [1]
- **Sustainability** defines how well application performs in long term from business perspective (mainly cost).
- **Testability** defines the extent of how easy or challenging it is to test an application.
- **Complexity** defines how complicated it is to understand the architecture, deployment cycle and implement changes.

Every software project consists of at least 6 phases: planning, design, development, testing, deployment and maintenance. I will focus on design because this is where the high-level decision about the architecture is made. Ideally, a software architect creates the concepts and designs for software and helps to turn those concepts into plans, just as an architect designs buildings. In smaller projects, the role of software architect will usually fall to the most experienced developer. While building architects are not usually concerned with how their ideas are implemented, a software architect is involved in all stages of the development process, not just the architecture, but any high-level decisions about the tools, coding standards or platforms to be used.

The architecture of the system is probably the most important decision to be made, as it influences all subsequent stages of development and is usually quite difficult to change afterwards. Interestingly, the choice of architecture is not only about meeting all the requirements of the project, but it must also fit the style of the organisation. There is an IT theory created by computer scientist/programmer Melvin Conway in 1967 which states: “Organisations that design systems are forced to produce designs that are copies of the communication structures of those organisations.”[2].And this theory makes a lot of sense. When faced with a difficult decision, people are always more likely to choose something they know well, rather than something that might be better but with which they have no experience. Also, not all designs may be compatible with our organisational structure. For example, a monolith with a 4-month release cycle won’t work for an early-stage startup that adds features every week and works in short iteration cycles. And vice versa. Having a microservices architecture, fast iteration cycles and the ability to

deploy as soon as a feature is ready is great, but when applied in an organisation with a complex hierarchical structure where simple changes take weeks to get approved, it does not even remotely exploit the benefits of the architecture.

The importance of choosing the right architecture for many software projects is underestimated and influenced by new shiny trends and fancy words in the IT industry, rather than being driven primarily by requirements, which can lead to unstable, inefficient and overpriced projects.

1.1 Monolith

Probably the most well known architecture praised by some, hated by others is the Monolith.Surprisingly, many people I know do not imagine a properly structured application, but rather 'Big ball of mud' [3] (haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle). Others refer to it as some kind of legacy system that should be eliminated as soon as possible. Of course, there is some truth in all of these statements. Monolithic architecture has been with us since the early days of software development, so there are plenty of legacy systems out there, but we need to understand that they are **legacy systems**. They were written decades ago, with architectural designs from a time when architectural research was in its early stage. To give some numbers - according to Google Scholar, 20 thousand articles were published about *Software Architecture* until the year 2000 [4] and from 2001 to 2023 it was over 244 thousand [5]. The progress in software architecture has been enormous in the last two decades, and new concepts for monolithic architectures have also been created.

In this section, I'm going to go into detail about what a monolith is, what it isn't, what new approaches are available, and try to clear up some common misconceptions. When I talk about monoliths, I am primarily referring to a unit of deployment[6]. *When all functionality in a system has to be deployed together, we consider it a monolith [6].* There are at least three types of monolithic systems that fit this description: the single-process system, the modular monolith, and the distributed monolith [7].

1.1.1 The Single process Monolith

The most common example of a system where all the code is deployed as a single process, as in Figure 1.1. There may be multiple instances of this process running for scalability or availability, but essentially all the code is packed into a single process. Typically, these single-process systems can be simple distributed systems in themselves, as they almost always end up reading data from or writing data to a database. [6]

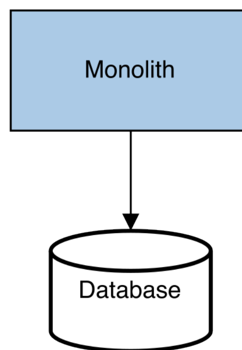
These single-process monoliths probably represent the vast majority of monolithic systems. There is no clear boundary between individual parts and the whole application is highly coupled [8], meaning that changing one part of the system inevitably affects other parts.

1.1.2 The Modular Monolith

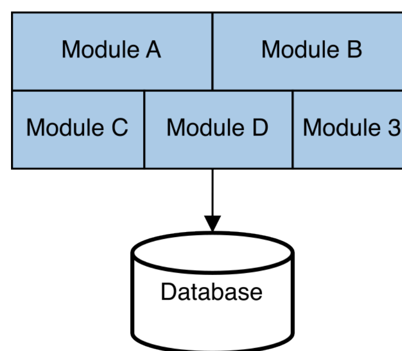
The modular monolith is a subset of the single-process monolith, where the single process is composed of distinct modules. Although each module can be worked on separately, deployment still requires all modules to be assembled into a single unit, as shown in Figure 1.2. [7]

This is a nice evolutionary step for monolithic systems. Well-defined module boundaries can allow a high degree of parallelism while avoiding the challenges associated with distributed microservice architecture and still having a simple deployment topology [7].

The biggest problem with this architecture is that although the logic is separated into modules, the storage is usually monolithic and represented by a database of relationships. This means that the separation is not effectively applied to the data, but only to the code.



■ **Figure 1.1** A single process monolith: all code is packaged into a single process. [6]



■ **Figure 1.2** In a modular monolith, the code inside the process is divided into modules. [7]

1.1.3 The Distributed Monolith

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable. [9]
>
> - Leslie Lamport

A distributed monolith is a system that consists of multiple services, but for some reason the entire system needs to be deployed together. A distributed monolith may well meet the definition of a service-oriented architecture, but all too often it fails to deliver on the promise of SOA. Distributed monoliths usually have all the disadvantages of a distributed system, and the disadvantages of a single-process monolith, without having enough upsides of either. [6]

Distributed monoliths typically arise in an environment where there has been insufficient focus on concepts such as information hiding and cohesion of business functionality, leading instead to highly coupled architectures where changes ripple across service boundaries and seemingly innocent changes that appear local in scope break other parts of the system [6]. In general, there is no reason to choose this distributed architecture over microservices, as it has many drawbacks and should be avoided, except when moving from monolith to microservices, where it may become an intermediate step [10].

I include this type of monolith for completeness, but since it is discouraged, I will not refer to it unless explicitly stated.

1.1.4 Service Oriented Architecture (SOA)

In the business world, creating solutions to automate the execution of business tasks makes a lot of sense. Throughout the history of IT, such solutions have been built using a common approach of identifying the business task to be automated, defining the business requirements and then building the solution logic [11]. This has been an accepted and proven approach to achieving positive business benefits, but it has had some negative sides:

- Repeatedly building “disposable applications” is not the perfect approach.
- The creation of new solution logic in a given enterprise typically results in a significant amount of redundant functionality[11].
- Applications built only with the automation of specific business task in mind, are generally not designed to integrate later with other applications well, resulting in a complex integration architecture.

All of those negative aspects listed above led to creation of Service-Oriented architecture with the idea of creating reusable services, requiring high-level of interoperability between service and numerous potential service consumers, with standardized contract, loosely coupled and composable, which requires to be standardized with cross-service data exchange.

Service-oriented architecture, or SOA, describes how to use service interfaces to make software components interoperable and reusable. By using an architectural pattern and common interface standards, services can be quickly integrated into new applications. As a result, the application developer is relieved of duties that previously required them to rewrite or replicate existing functionality or figure out how to link or ensure interoperability with it. [12]

In the early days, companies launched major organisation-wide initiatives to convert everything to a service-oriented architecture, which promised many benefits. However, the process was cumbersome and time-consuming, often requiring the company’s development team to re-architect all existing systems and design new applications according to new principles. A different way of approaching the problems of SOA was the Enterprise Service Bus (ESB), which can be implemented and deployed in a short time and acts as a central solution for integrations. The ESB approach was quickly adopted and is now used by virtually every company that has an SOA architecture. It allows existing legacy systems to be preserved, simply by exposing services from them via API and creating the integration through the ESB.

Every enterprise contains a service inventory, where all the services are located and from which the target application can be composed, as shown in Figure 1.3. The directly composable property should have mitigated the traditional perception of integration, but in reality it has just moved into the ESB. Once a significant part of the enterprise solution logic is represented by services in the inventory, the freedom to mix these services into infinite composition configurations to match whatever automation task comes our way [11].

A classification is used to indicate the reuse potential of the logic and how the service relates to the actual business logic. The common service models are: Task service, Entity service, Utility service and microservice (this is different microservice than we know today, in SOA it was used for small implementation specific service, which was not reusable). [11]

The complexity of implementation to achieve true reusability is huge and has noticeable performance drawbacks, not to mention high maintenance costs. In practice, this architecture has mostly been implemented in large existing companies that already had some legacy systems and combined with the ESB to create huge highly coupled monoliths. Today, everyone is moving away from this architecture, if they have not already done so, towards microservices (section 1.3), which have proven to be better for the job.

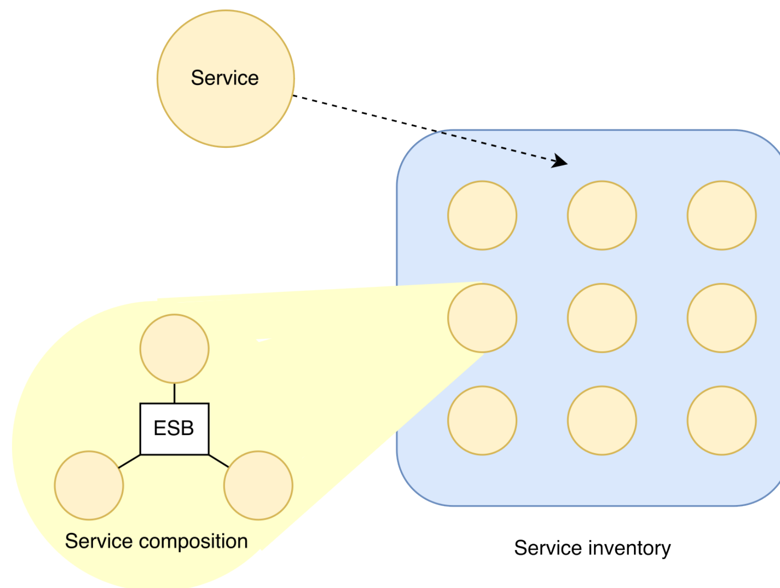


Figure 1.3 Services are delivered into service inventory (right) from which service composition are drawn (bottom).

1.1.5 Characteristics

1.1.5.1 Performance

Deploying as a single process gives a huge performance advantage over any kind of distributed system, because network communication is subject to the laws of physics and always adds latency. Inter-process communication, on the other hand, has the lowest possible latency, making the monolith theoretically the fastest architecture out there. This rule applies until the need to scale the application arises and vertical scaling is no longer an option. Since the whole system is deployed as a single unit, horizontal scaling means running multiple instances of the whole monolith, which in most cases is not very efficient, because the system is usually not evenly loaded, but some parts of the system are doing most of the work, and we still have those other parts taking up resources when we don't actually need them.

1.1.5.2 Maintainability

Maintenance is closely related to complexity. Usually, the more complex the application, the harder it is to maintain, and with monolithic high coupling it is very bad at it. A single change can have the power to affect the entire application, and maintaining legacy monoliths has become a nightmare for many developers, as even the smallest change requires extensive knowledge of the whole system. Although there are some positive sides, debugging and logging is much easier than with any kind of distributed system, so finding the problem is usually the easy step compared to creating the actual fix.

1.1.5.3 Sustainability

The maintenance costs of monolith are huge, as are the operational costs due to its inability to scale properly, so we have to have an oversized infrastructure, but also developers with extensive knowledge of the system and a well-defined testing process. Introducing any kind of architectural change is almost impossible and integrations have also proven to be complicated/problematic.

1.1.5.4 Testability

Since there are no boundaries in the classic monolith, it is almost impossible to test only certain parts of the system in isolation, because the whole system has to be present for it to work. So writing end-to-end tests is usually the answer, but it is much harder to write and even execute them, because they take much more time than just testing isolated parts.

With a modular monolith, testing is much easier because you can test only specific modules. The tests are smaller, which results in higher speed, and are easier to write because the developer can focus only on the specific part of the system.

1.1.5.5 Complexity

High coupling makes it very hard to do any modification, because it requires vast knowledge of the whole system and as the developer team grows they start to get into each other way, wanting to change the same piece of code. It also creates confusion about who owns the code and who makes the decisions [6].

The architecture itself (single process or distributed monolith) imposes no restrictions on how the application should be structured and gives the developer maximum freedom. In my experience, this has proved fatal in many projects as it is a heavy burden and if not properly designed at the outset and kept under control throughout the development cycle, it can very easily lead to a ‘big ball of mud’ [3].

High coupling is one of the reasons why the Modular Monolith was created. It retains all the positive characteristics of the single-process monolith, while reducing the coupling by defining boundaries within the monolith. It is then up to the developers to properly define and enforce these boundaries - this step is very important because less experienced developers often tend to simplify their work at the expense of the architecture.

Deployment is very simple compared to any kind of distributed system, as there is only one unit. The downside is the size of the unit, not even because of space (we have a pretty fast network and cheap disk space), but rather higher resource usage and much slower startup compared to a microservice.

On the other hand, when there are multiple teams working on a monolithic system, release planning requires close cooperation between all development teams to ensure that everything that goes into the build is production ready. This tends to lead to slow deployment cycles, where releases are deployed every few months, which does not cope well with today’s requirements to deploy much more frequently (e.g. Agile methodology [13]).

1.2 Modulith

‘Microservice architectures are all the rage these days, but what’s really important for long-term maintainability is modularity. It isn’t necessary to use a network boundary to create such modules.’ [14]

The word *Modulith* is combination of words ‘Modular’ and ‘Monolith’. In previous section was discussed Monolith architectures and one of them was *Modular monolith* 1.1.2, which basically enforces separation of logic into modules. Modulith is taking this modular approach even further by enforcing separation not just on logic, but on database/storage as well (see Figure 1.4). This way the modules are truly independent on each other, and they are in full control of its data, since now modules has to communicate with others via defined interfaces (this can be done for example directly via inter-process communication or messaging). Isolated, independent modules with interface through which they communicate formulate Modulith.

Modules being fully encapsulates give ability to even run different modules on different nodes/servers and communicate with others over the network, I call this **Hybrid Modulith**,

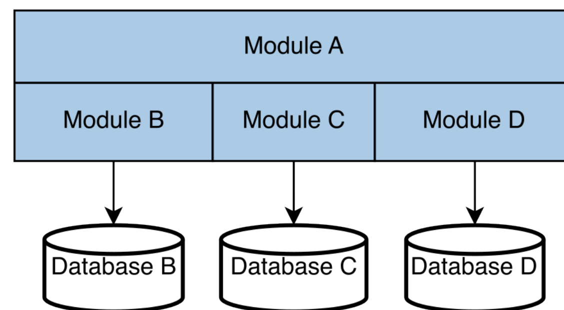


Figure 1.4 Mondulith architecture. Modules encapsulate logic and its own data.

and this is basically what Microservices are in its nature are. It is an enormous evolution compared to classical Monolith in terms of scaling, since this architecture offers ability to scale just parts of the systems (modules) instead of the whole application. Slow transition to Microservices (discussed in following Section 1.3) is natural step, once the need for scaling arise - just by moving required module into separate service we get a hybrid of microservices and modulith. So the whole application can be build using Modulith architecture, keeping all advantages of Microservices and in the same time removing the biggest disadvantage 'distributed system' with the ability to incrementally move to distributed system once it is absolutely necessary.

Modulith is a better structured Monolith with ability to scale. It still has the nature of Monolith, so it is deployed as one unit, which gives confidence of matching interfaces across modules, which is something that Microservices architecture is missing, easier debugging as it is not a distributed system and modularity of Microservice architecture for long-term maintainability. To improve deployment time for larger projects, we can deploy modules independently as modern languages generally support dynamic module loading/replacement at runtime, but this adds complexity as we need to ensure that all deployed modules are compatible and perform the update atomically.

1.2.1 Characteristics

1.2.1.1 Performance

Running as a single process, gives this architecture near same properties concerning performance as for Monolith 1.1.5.1 with a little drawback due to its added abstraction and isolation of data storage across modules, but the modularity being huge improvement in long-term maintainability. The Modulith architecture does not scale itself, but rather presume incremental transforming into microservices when the need arises. The right question is how much do we really need our architecture to scale. When starting a project, we can have some expectation on system load, but with how dynamically today project change from nearly from day to day, it usually starts as products A and finishes as product Z, so our presumptions on start will have to be adjusted as well. With that being sad, it is nearly impossible to guess what needs to be scaled beforehand, and rather have architecture, which allows us build fast with minimal overhead and with ability to scale once we actually need it. Also, if we look on modern hardware, we can find affordable up to 128 cores servers, which can easily run our wildest applications even without getting into distributed systems. Programming languages were adapted over the years as well: NodeJS with even loop can handle easily thousands of connections in single thread, Java recently got virtual threads (virtual threads), Golang has goroutines and Rust has asynchronous programming. All of those tools allows writing effective code when dealing with IO taks, which is what most of today applications are mostly made of.

Once we have to scale this architecture, some modules are moved into separated service,

heading towards microservices and the negative aspects of distributed system will start appear - mainly the network unreliability and latency, more discussed in Microservice section 1.3.3.1.

1.2.1.2 Maintainability

This is where the Monolith has been proved to be very problematic and the Microservice shines. Enforcing modularity on architecture level turned out to be essential in long-term maintenance. In my experience all projects usually begin as beautiful things and over time got messy. In my own opinion this is primarily caused by laziness of us developers. If there is some shortcut we can use, we will and say to ourselves: 'I will fix it later', which off course never happens. I am not saying this happens all the time, but there are times, when deadlines start breathing on our neck and forcing us to do things faster. So, when we remove some of those shortcuts as were in Monolith, which allowed to do anywhere anything and force modularity on architecture level as in Modulith, developers has no other choice then to do it properly, since there will be no other way. It does mean that some features will take few minutes or hours more, but this will be well invested time compared to searching for mysterious bugs in Monolith where much, much more time is spent.

1.2.1.3 Sustainability

Modularity provides great flexibility and allows for faster iterations to better adapt to changing business and requirements and architectural changes.

1.2.1.4 Testability

Testing is very similar to microservices, as the entire application is broken down into modules that can be tested independently. The big advantage even over microservices is the single-process nature, which allows testing using conventional tools and frameworks designed for monoliths.

1.2.1.5 Complexity

In terms of complexity the Modulith architecture sits somewhere between Monolith and Microservice. In its nature it is not a distributed system as Microservices, so there are no network issues and deployment process is as simple as for Monolith. Until we start running modules in separated service, we get partially distributed system, and the complexity arises. The advantages of Modulith is the incremental migration to distributed system and ability to decide whether it is actually needed, because with Microservices we have distributed system from the start even if we want/need it or not. The same applies to the implementation of transactions, until we start moving towards a distributed system the same transaction patterns can be implemented as were for Monolith. Distributed transactions will be discussed later in Microservice section 1.3.3.5.

1.3 Microservices

Many people see this architecture as a solution to every problem. What architecture for a new project? Microservices. Is your application slow? Turn it into microservices. Experiencing slow development? Microservices are the answer. To be fair, it really does change the game, and there are definitely projects where it is the best possible solution. Although, in my experience, people tend to use this architecture carelessly as the only correct solution, ignoring its drawbacks, and there are definitely projects where, for example, a monolith would be better suited to the task and save a lot of money and time. So what exactly is it?

Microservices are services that can be released independently and are modelled around a business domain. A service is a building block from which you can create a more complex

system. It encapsulates functionality and makes it available to other services over networks. A single microservice might represent inventory, another order processing and another shipping, but taken together they could make up an entire e-commerce system. Microservices is an architectural approach that aims to provide you with a variety of solutions to solve potential problems. [15]

They are a kind of service-oriented architecture, although they have strong opinions about the appropriate way to define service boundaries and emphasise the importance of independent deployability. One of the advantages they have is that they are technology agnostic. [15]

From a technology perspective, microservices expose the business capabilities they encapsulate through one or more network endpoints [16] (for example, a queue or a REST API [15], as shown in Figure 1.5). Microservices communicate with each other over these networks - making them a form of distributed system. They also encapsulate data storage and retrieval, exposing data through well-defined interfaces. Databases are thus hidden within the service boundary. [16]

Compared to the modulith discussed in section 1.2, microservices take the whole modular concept one step further. Instead of building the application primarily around modules, with the possibility of extracting the module into a separate service to scale it, they build the whole application from the ground up around services, with network boundaries fully embracing scalability. This is where I see the problem with building new applications with microservices architecture from the outset, when there is no idea about system load, performance issues or anything else. I would compare it to 'premature optimisation' in programming - why spend a lot of time building a super scalable system when we don't even know if we will ever need it?

What is the correct size of an individual microservice? The *micro* part of the word is misleading. Microservices were created primarily for the needs of large enterprises, and the original idea behind it was to have a dedicated team of developers behind each microservice. And this is something that is not possible in most companies with just a couple of developers. The practice I have seen in reality is that just a couple of developers are running dozens of microservices or even in one project I was a single developer managing over 20 microservices (more in chapter 2). Micro' simply means different things to big companies like Netflix or Amazon than it does to small/medium sized projects.

Building a microservice architecture requires a lot of planning to get the boundaries right for each service and to keep track of them. It is much harder to do major refactoring compared to monolithic architecture, although it should be easier due to service isolation. Unfortunately, in practice it is impossible to create something completely independent due to business requirements. Example: Suppose we have a microservice with some HTTP API that we need to separate. Other services are communicating with it through the API. We need to maintain compatibility for the API for a long time, because we most likely do not know exactly which services are using which API, so it is not safe to remove anything. So we split the service, the old service will have the full original API, the new service will only have the part of the API it can handle. For the old service, additional backwards compatibility logic needs to be implemented for those parts of the API that have been moved to the second service. And this compatibility needs to be maintained for an unspecified period of time - this depends on whether we have a strategy how to safely remove something and make sure it does not break other parts of the system. On the other hand, in Modulith we would just split the one module into two, and we very easily find all places which use the old functionality and update them accordingly - how to find them? IDE (basic function 'find usage'), compiler, static code analysis. The compiler automatically makes sure that all interfaces and implementations are compatible, and it does this for free out of the box. Something that is not easy to do in the microservices world.

1.3.1 Brief history

Dr. Peter Rodgers during a presentation in 2005 used the term 'Micro-web-Services' on topic of cloud computing. Rodgers promoted software components supporting micro-web-services.

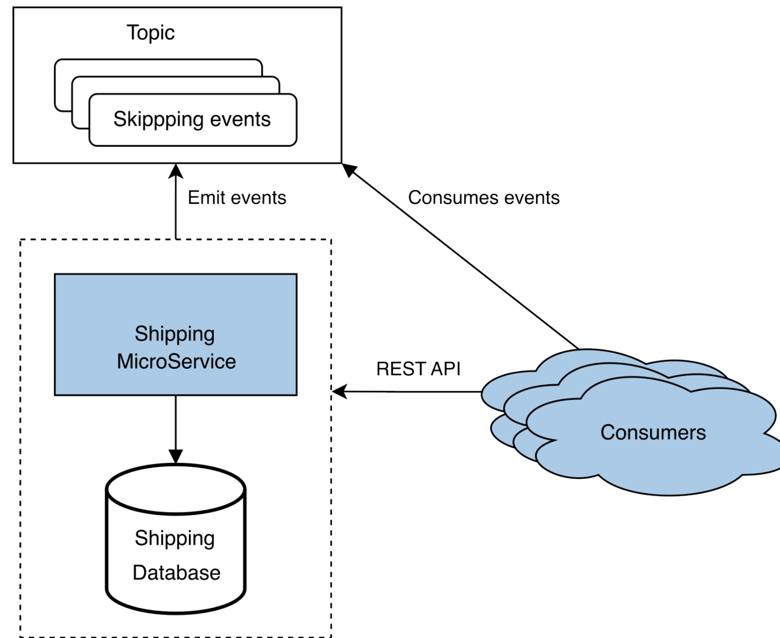


Figure 1.5 A microservice exposing its functionality over a REST API and a topic. [15]

The first usage of the term "microservices" is to believed to be first heard in May, 2011, in a workshop for software architects, where they use it to describe an architectural style several of them had recently explored. Later on in 2012, the term was formally adopted. They had been experimenting with building continuously deployed systems, while incorporating the DevOps philosophy. From there this form of the architecture quickly gained popularity. [17]

Netlix is a company that is considered a pioneer in the adoption of microservices, along with Amazon, which was more focused on cloud computing. Also, other large companies joined the ride like Uber, Etsy and many more. This architecture was the answer companies were looking for, due to its ability to offer agility, scalability and ability to adapt to changing business requirements. [18]

1.3.2 Not the proclaimed silver bullet

Over the last decade, microservices architecture has become so popular that very few people question whether it is the right architecture for the project at hand, and whether there might be something better for the job. One reason for this may be the incredible flood of articles on the web, with primers focusing on the positive aspects of microservices and how they can solve (almost) every problem, while not talking much about the possible negative aspects. It is important to note that most of the articles and success stories actually come from large companies with enormous resources (e.g. Netflix, Amazon, Coca-Cola) that none of the small/medium companies can match, while these smaller companies also face completely different challenges than the multinationals.

Microservices are very closely related to the modern concept of cloud computing, which has become a multi-billion market ($545 billion in 2022[19]) with the potential to bring huge savings, but also huge expenses if not used properly. And let's not pretend that these cloud service providers are not, of course, trying to convince us to convert our infrastructure and systems to new modern technologies in order to increase their profits. This is nothing new, as everyone is just trying to make money, but we should be aware of their intentions when they try to convince us to use their technologies - which in the case of microservices and their complexity of operation

and deployment will inevitably force us to use their services, but with monoliths we are usually capable of running everything ourselves.

To present some concrete data, let's look at some real-world examples where microservices were not considered to be the right solution.

▶ **Example 1.1** (Amazon Prime)**.** An example where microservices proved to be too costly a choice, and the entire application had to be migrated to a monolithic architecture for the system to be efficient, comes from Amazon. On 22 March (2023), Amazon's video streaming service called Prime Video published an article on their technology blog with the headline "Scaling the Prime Video audio/video monitoring service and reducing costs by 90%". Prime Video offers thousands of live streams to its customers. To ensure that customers receive content seamlessly, Prime Video set up a tool to monitor every stream that customers watched to identify any perceived quality issues. [20]

The initial version of the service consisted of distributed components orchestrated by *AWS Step Functions*. This would allow each component to scale independently. However, the way they used the components meant that they hit a hard scaling limit at around 5 % of the expected load. Also, the total cost of all the components was too high to use on a large scale. The two most costly operations were the orchestration workflow and passing data between distributed components. To address this, they moved all the components into a single process to keep the data transfer within the process memory, which also simplified the orchestration logic. With all operations now consolidated into a single process, they were able to rely on scalable Amazon compute (EC2) and container (ECS) instances for deployment. [20]

Conceptually, the high-level architecture remains the same. They still have exactly the same components as in the original design (media conversion, detectors, orchestration). This allowed them to reuse a lot of code and migrate quickly. Originally, they could scale multiple detectors horizontally because each one ran as a separate microservice. However, in the new approach, the number of detectors could only scale vertically, as they all ran within the same instance, and this would quickly exceed the capacity of a single instance. This limitation was overcome by running multiple instances and implementing a lightweight orchestration layer to distribute customer requests. Overall, the migration from a distributed microservices architecture to a monolithic one enabled them to reduce infrastructure costs by over 90 % and increase scalability. [20]

▶ **Example 1.2** (Shopify)**.** An example of a company that still successfully operates a monolithic system with thousands of developers is Shopify. It is a complete commerce platform that allows businesses to build an online store, market to customers and accept payments. The article[21] with the most insight into their architecture is from 16 Sep 2020, and they regularly write more insight stories about their Monolith on their *Shopify Engineering*[22] website.

They have a massive monolith written in the Ruby on Rails framework, its core alone has over 2.8 million lines of Ruby code. This is one of the oldest and largest Rails codebases on the planet and has been in continuous development since at least 2006. Rails doesn't provide patterns or tools for managing the inherent complexity and adding features in a structured, well-bounded way. That's why, in 2017, Shopify set up a team to investigate how to make its Rails monoliths more modular. The aim was to make it easier for them to scale to ever-increasing system capabilities and complexity by creating smaller, independent units of code, which they called components. The added constraints on how they wrote code triggered deep software design discussions across the organisation. This led to a shift in the mindset of developers towards a greater focus on modular design. Clearly defined ownership of parts of the code base was a key factor in the successful transition. [21]

The initial focus was on building a clean public interface around each component to hide the internals. The expectation was that changing the internals of one component wouldn't break other components, and it would be easier to understand the behaviour of a component in isolation. They had to balance the encapsulation with the dependency graph to avoid circular

dependencies, which are very risky because changing any component in the chain can break all the other components. Various techniques such as control inversion and publish/subscribe mechanisms were introduced to help minimise relationships and reduce coupling. In the end they ended up with 37 components in the main monolith, and they are very deliberate about splitting functionality into separate services due to the overall complexity of a distributed system of services. [21]

► Note. Netflix is one of the most cited companies for its microservices architecture. What is not so well known is that the actual migration from monolithic system, when they started to have problems with performance and scaling, was in 2009. At that point they had been using Monolith for 10 years and had over 11 million [23] paying subscribers by that time. No one knows what would happen if they started the original architecture with something else, maybe it would work, but definitely the Monolith worked well until it just didn't fit their needs anymore.

1.3.3 Characteristics

1.3.3.1 Performance

Because microservices are a distributed system, they inevitably have higher latency than monolithic systems, but they bring many benefits that can easily outweigh this downside. The biggest is scalability. Microservices are very small units that can usually start up in a few seconds, so they can scale based on the current workload, taking full advantage of cloud computing and pay-per-use pricing, rather than having an oversized infrastructure. On the other hand, scaling does not work out of the box and requires a system to support it. Also, it is not possible to scale infinitely and some bottleneck will stand out, usually the database.

1.3.3.2 Maintainability

Distributed systems are much harder to maintain because of the added complexity. Even something as simple as investigating a problem requires collecting logs from multiple servers, aggregating them and having a tool to effectively search through them. Or even better, use distributed tracing, which can generate huge amounts of data even on simple systems. Changes in microservices have an isolated effect within the microservice and can be deployed in a matter of hours (or even minutes).

1.3.3.3 Sustainability

The isolated microservices provide great flexibility and allow for faster iterations (even faster than Modulith) to better adapt to changing business needs and architectural changes.

1.3.3.4 Testability

In theory testing should be breeze, but in really not so much, because services are usually part of a larger business-logic dependent on other services to complete the logic. Practices for service-testing (e.g. mocks, API-contract testing) are known, but they are very complex and costly to implement. For comparison in monoliths we have many proved testing strategies including end-to-end with many great frameworks/libraries available. [24]

Service can often become inaccessible, due to issue on many layers of network or security operated by an orchestrating software. These kinds of problems are hard to simulate in-order to prevent faults from reaching production.

1.3.3.5 Complexity

Breaking whole system into isolated 'micro' pieces (services) makes it easier for developers to implement new features and work independently, but it definitely makes the whole system more complicated, because of added network communication and more complicated deployment. Network communication is required for inter-service communication and also for outside world, and there is a lot we can choose from. Starting from classical patterns like request-response or more data driven approaches like observer or publish-subscribe pattern.

Managing deployment of microservices is a big task and over the years lot of automation tools were created. Today standard way on how to deploy applications with rise of Docker became containers, due to its self-containement, isolation and ability to run absolutely anywhere [25]. Container Orchestration make deployment and maintenance much easier thanks to automatization - examples: Kubernetes[26], Nomad[27]. It takes care of whole lifecycle of the application and even auto-scaling. Unfortunately, managing those tools is a big challenge on its own, but thankfully nowadays we can get fully managed orchestrators 'as a Service', so we can fully focus on building our system and not on infrastructure.

Distributed transactions have the same processing requirements as regular database transactions, but they must be managed across multiple resources, making them more challenging to implement. When ACID transactions are required, the 2PC [28] (two-phase commit) architecture with central transaction coordinator is a one solution. On the other, hand if eventual consistency is enough, there is SAGA pattern [29], which works sequentially.

Language and technology agnosticism is another advantage presented around microservices. Personally, I see this as a negative for small teams, where few developers manage dozens of microservices and they would have to constantly change context. But it is a great feature for large teams, where ideally a team has only one or a few microservices. It gives a lot of freedom and developers like to play with every new shiny piece of software/technology, which (if unchecked) can lead to a technology jungle that no one understands and no one can maintain. The same goes for languages. While I am a strong advocate of always using a language for the right task, it must be well argued and everyone on the team should be familiar with it, because in small teams there is usually shared code ownership.

1.4 Summary matrix

Following Table 1.6 contains summary of previous sections discussing multiple architectures.

Aspect	**Monolithic**	**Modulith**	**Hybrid Modulith**	**Microservices**
Structure	Single, highly coupled application.	Single, loosely coupled application	Multiple loosely coupled Moduliths.	Many small, independent services.
Communication	Direct function calls.	Direct function API calls.	Network communication.	Lof of network communication.
Scalability	Add instance of whole application.	Add instance of whole application.	Add instance of individual Modulith.	Add instance of individual microservice.
Architecture overhead	Minimal	Minimal	Moderate	Higher
Maintainability	Modification can affect the entire application	Modification in modules has isolated effects. *	Modifications in module have isolated effect. *	Modifications in microservice have isolated effect. *
Deployment	Single unit.	Single unit or independent deployment of modules.	Independent deployment of Moduliths or individual modules.	Independent deployment of microservices.
Collaboration	Everyone works on one codebase.	Everyone works on one codebase.	Teams collaborate on individual Moduliths.	Teams collaborate on individual microservices.
Complexity	Higher due to highly coupled components	Lower complexity with highly decoupled modules.	Moderate complexity with decoupled modules (lightly fragmented distributed system).	Higher complexity with highly decoupled microservices (highly fragmented distributed system).

Figure 1.6 Overview of architectures. *Modifications to API always affects the whole application.

Chapter 2

Financial well-being project

In this chapter, I'll describe one of the projects I worked on that was built from the ground up using a microservices architecture. I will share my experiences from a backend developer's point of view on what it means to work alone and in a team on a microservices architecture. What are the advantages and disadvantages for developers during the whole application cycle (planning, implementation, deployment, maintenance).

2.1 Introduction to the project

At the time of writing, I have been working for the company for over a year on their financial wellbeing platform. The purpose of the platform is to help users connect with coaches and give them tools to educate themselves, get their finances under control and plan for the future.

This project has been in development for over 4 years with only a handful of developers. The backend is written using microservices architecture in Rust and there are 2 frontend applications: native iOS and web. There are also two additional web frontends for administration. Admin portal is used by financial advisors to communicate with users and also by support/admin users. The latter is the customer portal, available to our customers (employers) to see their employees' interactions with our platform.

Over the years the platform has gained a lot of features, so I will just highlight the main ones to give you an idea of what the platform does. I will describe the processes using user journeys:

► **Example 2.1** (Sign Up)**.** Joe, an employee of company Z, receives an email about gaining access to the platform, which his employer has purchased as a benefit. Joe opens a web browser and goes to the registration page. He enters his email and password and goes through the onboarding process (a few basic questions about his financial experience and goals). Once registration is complete, he lands on the home page.

► **Example 2.2** (Advisor)**.** Immediately after signing up, Joe is presented with the name of his personal financial advisor. He can send him a message via chat, or better still, schedule a phone call so they can meet face-to-face, get to know Joe's expectations, review his finances and set some goals.

► **Example 2.3** (Analyse)**.** Meanwhile, as Joe waits for his first meeting with his adviser, the platform offers to connect his bank accounts so it can give him some insight into his spending. In the UK, there is an open banking standard that allows him to give the application access to his accounts and transactions in just 3 steps. Once connected, Joe is presented with graphs showing how much he spends each month in each category. He can also set goals (e.g. wedding or buying a house), link them to accounts and it will automatically track any progress (savings).

▶ **Example 2.4** (Education). On the Learning page, Joe can read thousands of financial articles written by our financial advisors or watch recordings of past webinars. He can also view and register for upcoming webinars.

2.2 Infrastructure

The infrastructure is based on HashiCorp products (HashiStack). A cloud provider is used for VM provisioning and networking, but we manage the rest ourselves via custom Terraform scripts (IaaS). What are the key infrastructure components:

- *Nomad is a simple and flexible scheduler and orchestrator for deploying and managing containers. [27]*
- *Consul is a service networking solution that automates network configuration, service discovery and secure connectivity. [30]*
- *Vault secures, stores and tightly controls access to tokens, passwords, certificates, API keys and other secrets critical to modern computing. [31]*

I don't know what the reason was for going with the HashiCorp stack, but looking at the experience with it after a year, I'm amazed at how well the components integrate, and the setup/management of the whole cluster was a breeze compared to Kubernetes (K8s). There are now many managed K8s clusters available, but not a single managed Nomad cluster provider that I could find (there was an unmanaged.io project, but it seems to be down). I think the reason for this is simple - Nomad is just easy to run and manage. I'm not condemning K8s, I'm sure the complexity is there for a reason, but just for this project Nomad was more than enough and much easier to work with. During the year we had 2 incidents in production. The first was caused by a weird cluster state, which was resolved by rebooting, and the second was caused by our cloud provider having network issues. As we had no one dedicated to infrastructure and everything was managed by the BE developers, we just needed something that was easy to use and would work. Nomad did exactly that without adding any complexity that we did not need and that K8s would most likely add.

Even though this is a relatively simple microservices infrastructure, it still consists of a lot of components that new developers needed some time to get used to. A few examples:

1. The API gateway was created using a proxy called *fabio*, which has it's own way of defining path mapping.
2. Nomad has it's own job definition language called HCL.
3. The inter-service communication needs to be protected because it is over the network. So additional secrets handling and service discovery had to be done before each inter-service call.
4. Changing deployment properties requires vast knowledge of devops and editing quite complex pipelines.

None of the above examples of obstacles exist when working with monoliths.

2.3 Microservices architecture

The whole platform consists of 26 microservices, 4 of which are backend for frontend. Each microservice exposes its API via HTTP protocol for inter-service communication as well as to the outside world and API Gateway is created via reverse proxy called *fabio*. For background

tasks, a messaging system with queues is used, specifically Amazon's SNS for sending messages and SQS for receiving them.

Git is used for versioning with one repository per microservice. I see some advantages in having separate repositories compared to monorepo. Firstly, pull requests are better organised and compact - changes to a microservice should be isolated, and having Monorepo allows code in multiple microservices to be touched, and it would be up to the code reviewer to notice. It also forces developers to think of changes as more isolated, since they can only change a single microservice in a single repository - they have to switch to another to make the next changes. The same flow could be enforced by git hooks, for example, but this way it requires no configuration. On the other hand, it is a bit harder for newcomers to simply download all repositories, as 26 is quite a lot and requires some structure to be added to local repositories for easier navigation.

Finding the right size of microservices is a difficult task. In this project, the domain driver approach was used to find 'small enough pieces' (this was an approximation) and map them into microservices. Some have well-defined boundaries, some have a gray area, few are overgrown and should have been split because maintenance has become much harder with increasing complexity. A few examples:

- *Financial Account* holds information about bank accounts for users. Either manually created accounts or those that are connected with bank.
- *Transaction* stores financial transactions.
- *Categorisation* contains pure logic that categorises transactions into different categories.
- *Orchestrator* is used for scheduled tasks that trigger complex flows involving multiple microservices, such as updating cached accounts and Open Banking transactions, and starts the post-processing.
- *Advisor* was designed to hold the logic and data around advisors and their relationship with users. Later, chat functionality was added, and as it started with just basic messaging between advisors and users, the unfortunate decision was made to include everything around messaging here as well. Over time, this functionality has been extended and this microservice has become the most bloated of all.

From the examples above, it is clear that microservices have varying scopes/boundaries. Even if we have the best possible design at the start, over time, features will change and this will inevitably lead to an uneven distribution of boundaries. This does not mean that there is a problem with the design, as our system might just be adapting to our changing business needs, shifting goals, and our design needs to adapt as well. There are usually two outcomes. The first is a proper adaptation of the design, once we realise that the change is needed, and with it a possible refactoring/split/merge of microservices. The second option is what happened to the Advisor service. Over time, it took on more and more functionality, even outside its scope, and when someone later realised the mistake, the amount of work required to make the necessary changes was simply too much to invest at the time.

2.4 Characteristics

This section describes the project along the same lines as the architectures compared in the chapter 1.

2.4.1 Performance

In terms of performance, there were only two services that had noticeable CPU usage and those were processing thousands of financial transactions every time the user opened the analysis page,

the rest was mainly waiting for the database to read or write data. Despite this, we had very linear traffic and each service was only running in two instances with no autoscaling and only a few hundred MHz of CPU allocated to it, so it was almost like Monolith with two instances, with the exception that even internal requests were load balanced and not just external as with Monolith. So in this case, it would easily run as a monolithic system without any problems, with plenty of room to grow vertically: with 26 microservices, each would have 500ḢMz of CPU and 100ṀB of memory allocated. The total resource requirement for a single monolithic instance would be 5 CPU cores (2ĠHz per core) and 1ĠB of memory.

2.4.2 Maintainability

The idea of having lots of micro independent services is great from a scaling and rapid deployment perspective, but as with everything, there are negatives. Each service defines its own interface, in this case as a Swagger definition, and is kept in sync with the code manually, which inevitably leads to problems. There are some automated solutions that either generate the OpenApi definition from the code or vice versa. Unfortunately, it has some limitations, not every language has support for it, and I have not seen any project that actually uses it and relies on it completely. These outdated contracts have led to confusion for front-end developers, an inability to rely on it, and in practice, every time a front-end developer needed detailed behaviour about an endpoint, he had to contact a back-end developer who would investigate the behaviour directly from the code. Today, gRPC technology would probably be better for inter-service communication, as it has support for every major language and can automatically generate either server or client implementations from schema.

2.4.3 Sustainability

Defined contracts, whether via OpenApi or any other technology, ideally need to be maintained indefinitely, which is of course not possible. Even with the best design in mind, requirements will change, features will become obsolete, and maintaining each obsolete/old/unused feature will take a lot of time. In monolithic/modular statically typed systems, when someone changes the interface/contract, the compiler gives instant feedback as to whether these changes can be made, or whether there are conflicts, and where exactly. In the world of microservices, this is a much more complex issue. Taken to the extreme, each service implements its own client to communicate with all dependent microservices. When this happens, as it did in this project, how can you tell if even a specific change as small as removing a single field on the contract will affect other microservices, if any? Perhaps using an advanced static code analysis tool for this specific purpose would give an idea, but I doubt it would be reliable. Another option I took after seeing this, also code duplication with this approach, was to publish the client implementation with each contract change. That way, other microservices could simply add the client library as a dependency instead of implementing their own. Tracing dependencies between services was now as simple as checking dependencies and looking for client libraries. Now at least the incompatible changes to contracts are at least somewhat possible, as a new version of the client library can be released and each microservice can be rebuilt (e.g. as part of CI) with the updated client library to see its impact.

2.4.4 Testability

The worst part on this project was, and most likely is with every microservices project, the inability to properly check if all the contracts for each service are fully compatible, or if there is simply a typo somewhere, or if during refactoring a service was simply left out, which actually happened here from time to time. Even though we had the code coverage, the tests were still

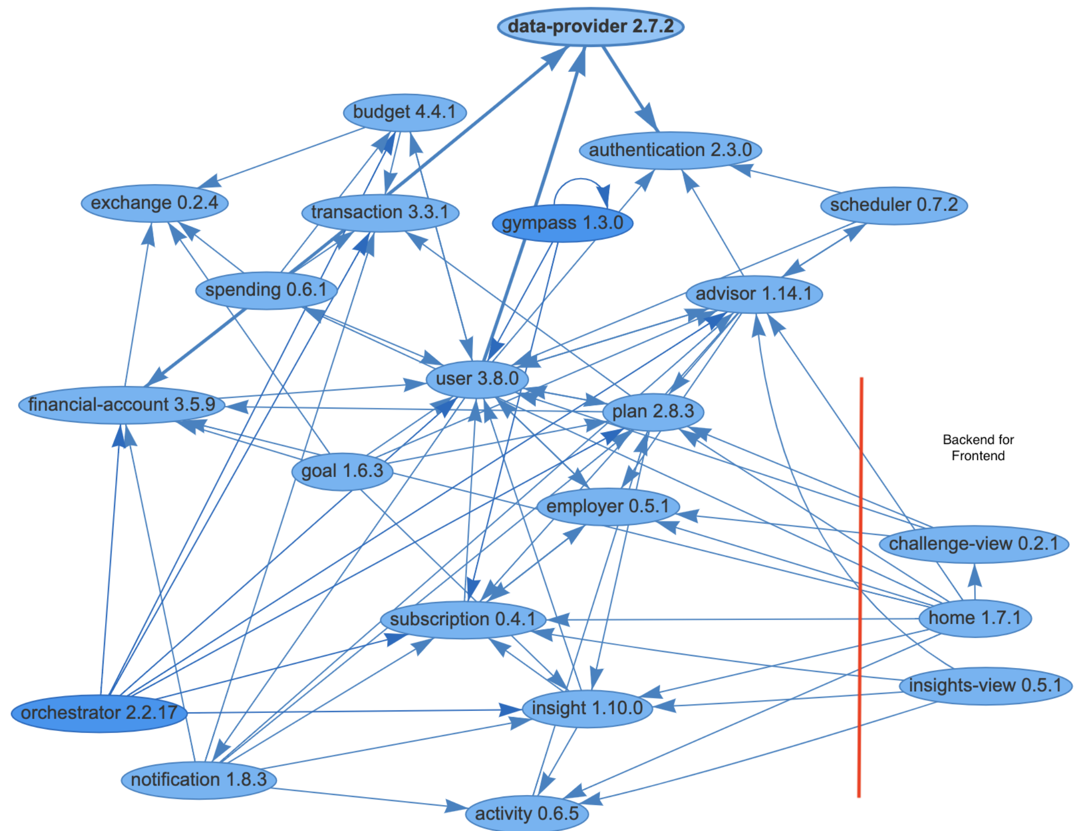
expecting the older version of the contract, so the problem would show up once it was running in the real environment. At best, we would catch it in development, at worst it would show up in production after release.

2.4.5 Complexity

Leveraging our existing code convention across services, I was able to generate a dependency graph for the whole system via static analysis, see graph 2.1 (something that was not available and no one paid attention to). This is a real example of what can happen to a microservices architecture after 3 years of development, and this graph does not include communication via async messages. It is not easy to say whether the state looks bad or not, but I would lean towards bad after seeing the graph. If this were the module dependency graph of Modulith, it would take some time to make changes, but once done and the compiler was happy, it would be done. But if someone wants to do these big kind of changes like change contracts (internal just between services) in microservices it is much more work having to change contract, client implementation, propagate the change to every service where it is used. Just to see where it is used is a task on its own compared to a single process system where every IDE is fully capable of showing you in a matter of seconds every single usage in the codebase.

2.5 Summary

The chapter contains a detailed description of my personal experience with a microservices project, highlighting the added complexity of the chosen microservices architecture and how intertwined individual services can become, despite the architecture promising low coupling. Because one thing is the “feature” of the architecture, and quite another is how well it is actually implemented and maintained over the years of development.



■ **Figure 2.1** Inter-microservice dependency graph of the backend after about 3 years of development. Each node represents a microservice and each arrow represents a direct dependency on another microservice.

Chapter 3

Proof of concept

So far, three different types of architectural patterns have been introduced. In this chapter I will create an implementation of a simple application using all 3 architectural patterns discussed (Monolith, Mudulith, Microservice). I will describe the architectural implications in terms of internal structure, database design, scalability and thoroughly measure performance of every application.

3.1 Application

This is an application example of a real world system consisting of HTTP API, database queries and business logic. It has been designed to be easy to understand and to solve a problem known to everyone: orders. Each application exposes the following HTTP API:

- **Get /item** - Retrieve all existing items.
- **Get /item/{itemId}** - Retrieve an existing item specified by id.
- **Post /cart/items/{itemId}** - Add item to shopping cart.
- **Post /order/create** - Create an order from items in shopping cart.
- **Get /order/{orderId}** - Retrieve order specified by id.
- **Get /invoice/{invoiceId}** - Retrieve invoice specified by id.
- **Get /payment/{paymentId}** - Retrieve payment specified by id.
- **Post /payment/invoice/{invoiceId}** - Pay for invoice specified by id.

The Client of the application can view items, add them to his shopping cart, place an order, retrieve an invoice and pay for it. Like most of the applications today, there are mainly operations querying data or saving data into database. The activity flow is described on Figure 3.1. First, a session is created for the client, then the client loads items and adds everything he wants to his shopping cart. Later the client places an order, an invoice is generated in the background and the client can cancel the order or pay for it. To add more CPU intensive tasks, invoice generates PDF and also calculates 41st Fibonacci number.

Candidates for the implementation language were Rust and Golang due to its minimal runtime (minimal impact on benchmarking). The winner is Golang because it's easier to use, I have good personal experience with it, and it has better support for the Open Tracing project, which is used

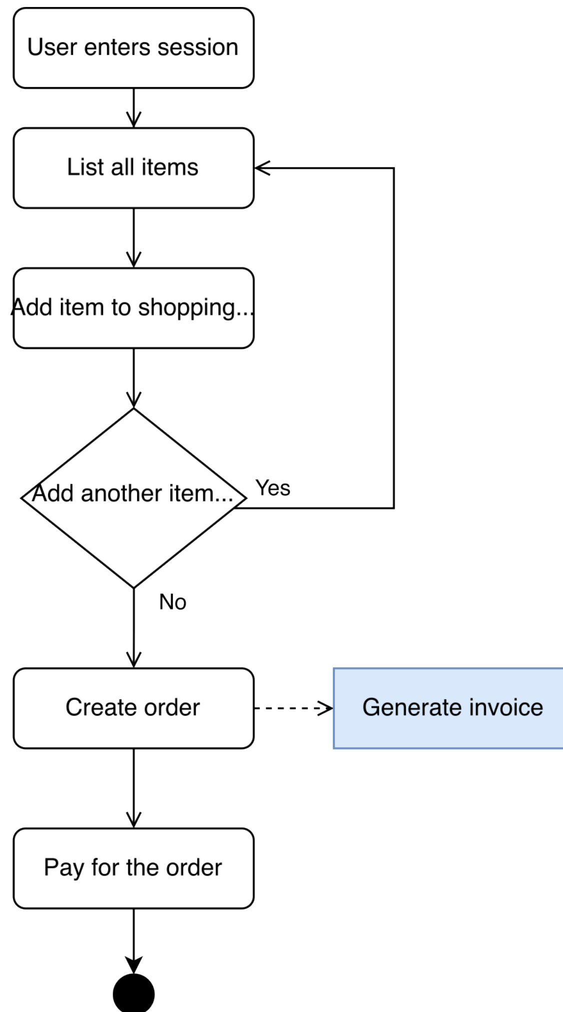


Figure 3.1 Diagram describes client flow of application.

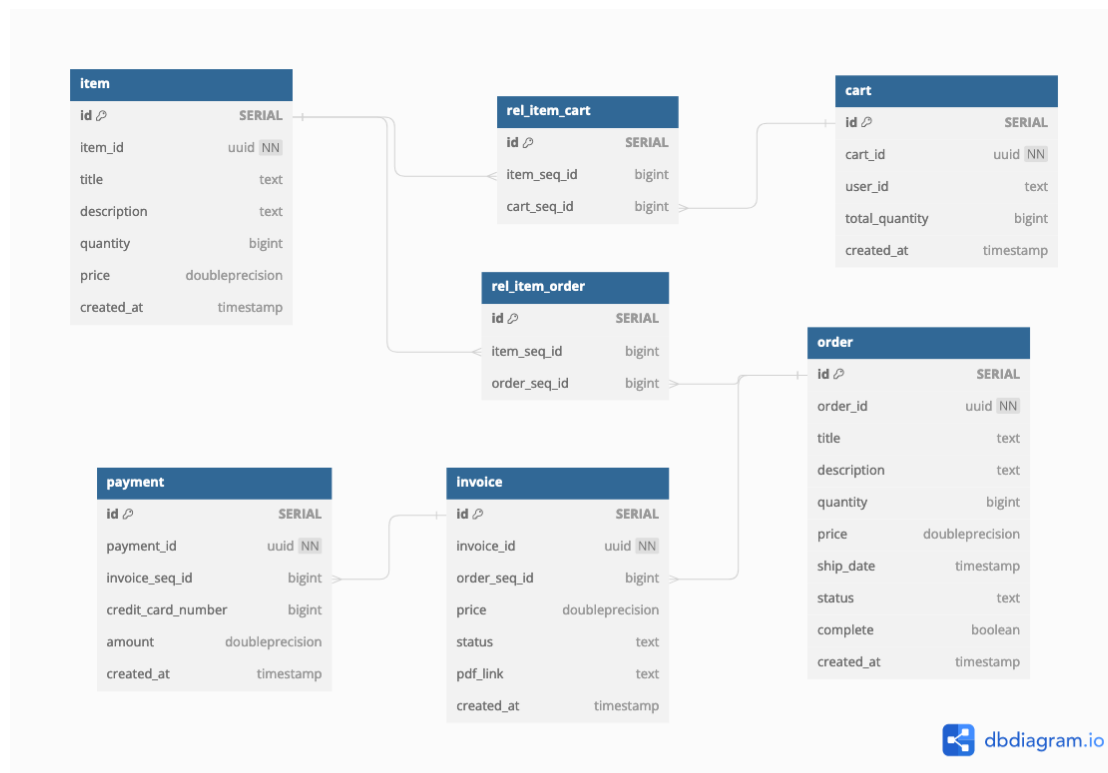
for monitoring from within the application during benchmarking. All applications are based on *gorilla/mux* [32] for http server + request routing library and *Bun* [33] as a lightweight ORM.

PostgreSQL was chosen as the data store because SQL databases are more widely known than NoSQL, so it should be easier for anyone to understand the design. It was chosen because of its current popularity and my preference over MySQL.

The application needs to be able to run in multiple instances so that it can be properly benchmarked and scaled later. In order to run the application in multiple instances, a container orchestrator will need to be used. Instead of adding unnecessary complexity when using Kubernetes, I decided to use *docker compose* to manage containers, as I do not plan to run the application on multiple nodes. All requests are sent to Nginx, which acts as both a reverse proxy and a load balancer.

3.2 Monolith example

This example is implemented as a monolithic system. Basically it is a three tier application: presentation (HTTP API), application and data (database). Figure 3.2 describes the internal



■ **Figure 3.2** Database schema displaying tables and relations of Monolithic example.

structure of the application packages. *Routes* package contains path mapping for endpoint handlers defined in *endpoints*. Handlers contain simple logic definition, more complex logic is in *services* and data manipulation around the database is in *database*.

3.2.1 Database

Due to the nature of Monolith as a unified system, all data resides within a single database, making full use of constraints and referential integrity, thus ensuring data consistency. Complete database schema is shown on diagram 3.2 consisting of 7 tables, five of which are entity tables and two are entity relationship mapping. The application will have a database pool consisting of a maximum of 8 connections.

3.3 Modulith example

This is an example of using the modular monolith approach. The original monolithic application has been split into several smaller monoliths called *modules*. The size of each module depends on the specificity of the project. In this case, it is almost equivalent to one module per database entity, except for the shopping cart and items, which have been merged into a single module to demonstrate the possibility of having modules with a larger volume. Package schema and dependencies are shown in diagram 3.3c.

Each module has the same internal structure as the original Monolith plus exposes its API with an interface (Diagram 3.3b). Modules encapsulate their own data storage, and there should be no cross-module table constraints in the database, although this is possible, it doesn't make sense from a logical separation perspective. It should be possible for each module to use a

different database and even a different database technology. In the case of the largest module, which contains the shopping cart and items, foreign key constraints are preserved because it resides within a single module.

Modules can use API of other modules, although this should be limited as much as possible to keep coupling low. The dependency on other modules is defined through the use of interfaces, and the actual implementation can either be automatically injected using the IoC approach, or as in this case, just define a top level module that takes care of initialising individual modules and spinning up a single HTTP server.

Scaling Modulith once we have identified the bottleneck packages is very easy. Since each module is essentially a small monolith, it can be moved into service and run independently. It just needs to expose its API over the network, such as gRPC or just a simple HTTP API. This network exposure can be generated automatically if all objects in the interface are serialisable. Later client instances will be passed to all dependent modules and from the point of view of other modules nothing changes, only now the underlying communication will not be inter-process communication but network communication. The implementation could even be fully automated in a declarative way. The package can be moved to a separate service and scaled, simply by changing the configuration.

In this particular example, the implementation to expose the capabilities defined in the invoice interface was created manually by defining the following additional HTTP endpoints and thus implementing a client to satisfy the existing interface.

- **Post /invoice** - Generate invoice for order specified in body.
- **Patch /invoice/{invoiceId}** - Update invoice specified by id.

The Modulith application has been extended to run in 3 modes, depending on the value of environment variables. In the first mode it runs as a single process. In the second mode it runs as a single process without invoice implementation and uses the invoice client. In the third mode it runs only the invoice implementation and this can be scaled to many instances to improve performance. Load balancing of requests to the invoice service instances is done using Nginx.

3.3.1 Database

Each module encapsulates its own tables, completely independent of the rest. The tables are the same as for the monolithic example on diagram 3.2, but the relations between modules have been removed. The schema consists of four partitions with relations preserved within the partition: the first partition consists of the invoice table, the second of the payment table, the third of the order table with rel_item_order and the last of the item, cart and rel_item_cart tables. Each module will have its own database pool with a maximum of 2 connections (8 in total).

3.4 Microservices example

In this example, the Modulith implementation has been further split into 5 microservices: Item, Shopping Cart, Invoice, Order and Payment. Modulith originally contained 4 packages, 1 of which combined the logic around Shopping Cart and Items, so it was split into two to make the scope more consistent across the microservices. The dependency graph between services is shown in Figure 3.3d.

Splitting Modulith into services requires exposing additional functionality over the network, as Modulith modules communicate with other modules via inter-process communication. As a result, additional endpoints (compared to the monolith) and clients had to be created to satisfy the defined interfaces.

- **Post /invoice** - Generate invoice for order specified in body.

- **Patch /invoice/{invoiceId}** - Update invoice specified by id.
- **Delete /cart/{cartId}** - Remote cart specified by id.
- **Get /cart/{cartId}/item/id** - Retrieve item ids within cart.
- **Get /cart/user/{userId}** - Retrieve cart for user specified by user id.

Each microservice runs an HTTP server and exposes HTTP API for its internal API to be used by other modules and also public HTTP API to be used by clients. With microservices it is common to use some kind of service discovery and let services communicate directly with each other. In this example, to keep things simple, nginx was used as a load balancer, acting as an intermediary and directing requests to the right services by path matching.

The whole application is compiled into a single binary, but in production it would most likely be compiled into binaries per service. Which service is started is controlled by an environment variable.

3.4.1 Database

Each microservice encapsulates its own data. The database tables are the same as for the monolithic example on diagram 3.2, but the constraints outside the microservice scope have been removed. Database partitions are the same as for Modulith, only item table has been separated into its own service after splitting one package into two separate: cart and item. Each microservice will have its own database pool with a maximum of 2 connections (10 in total).

3.5 Benchmark methodology

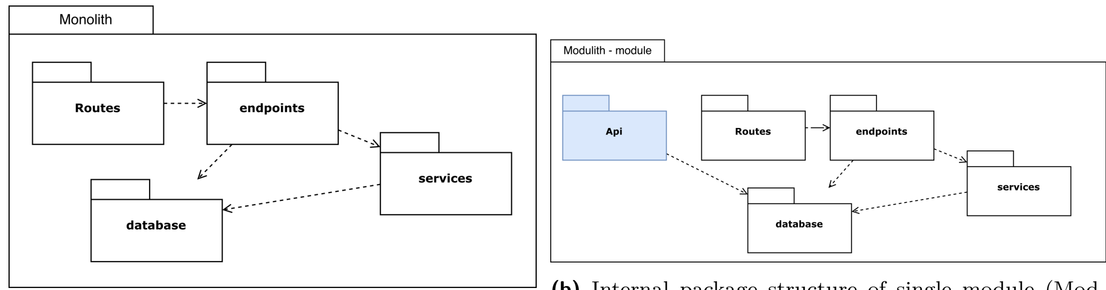
There are two types of scenarios that will be measured to get inside the bottlenecks of applications from both CPU intensive and IO intensive perspective. The benchmarking is performed using the K6 load testing tool, which acts as an HTTP client sending requests defined by the scenario.

Benchmarking will be done for following test scenarios with various configurations:

- Performance scenario - focusing on even load of whole application testing IO and CPU intensive operations.
- Latency scenario - focusing primarily on IO operations and handling a lot of fast requests.

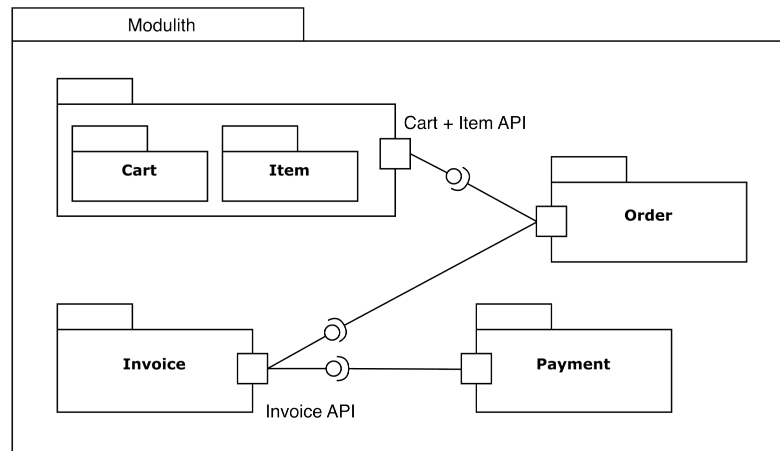
All benchmarks results will originate from running 10 parallel tests, each 100 iterations with following configuration.

- Golang 1.21.3 - programming language used for applications
- PostgreSQL 14 - database
- Docker 24.0.5 - execution environment
- OpenTelemetry - monitoring from within application
- k6 - load testing tool
- Hardware - Macbook Air 13” with M2, 16 GB Ram, 6 CPU cores assigned to Docker
- Every service instance has following assigned resources unless specified otherwise: 0.5 CPU and 50 MB of Ram

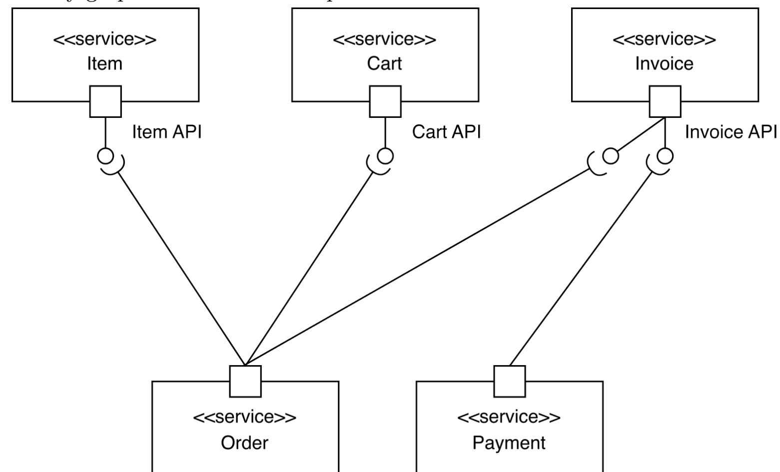


(a) Internal package structure of Monolith example.

(b) Internal package structure of single module (Modulith example).

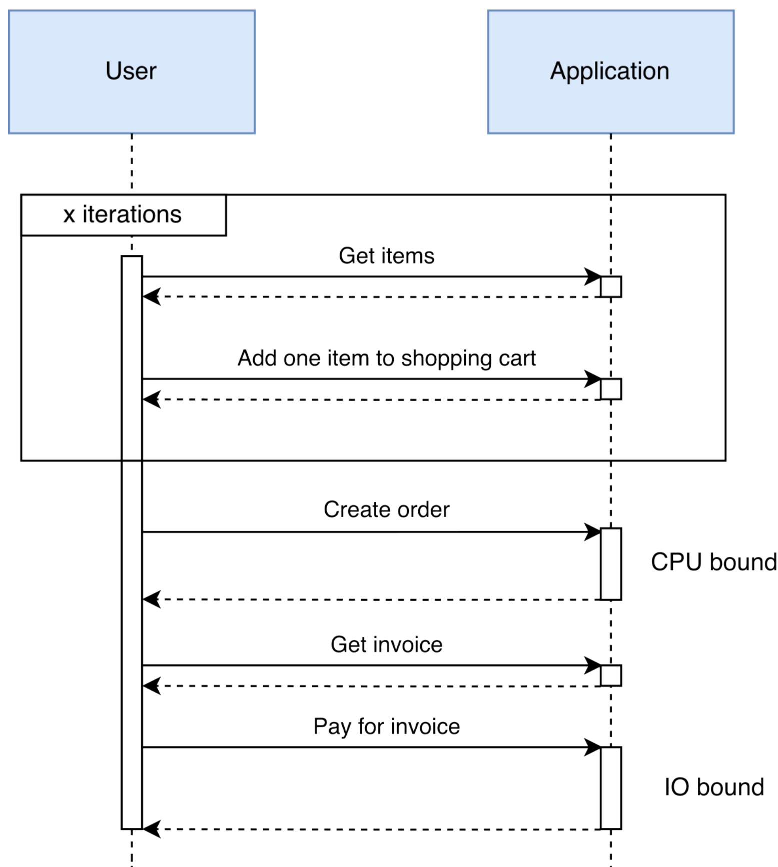


(c) Module dependency graph of Modulith example.



(d) Dependency graph between services of Microservices example.

Figure 3.3 Architecture diagrams for application examples build with architectures: Monolith, Modulith and Microservices.



■ **Figure 3.4** Benchmark flow diagram.

3.6 Benchmark results

3.6.1 Performance scenario

The first scenario consists of going through the whole application flow as defined on Figure 3.4, which should represent evenly distributed load on the whole system. Client load list of hundred items and adds one random into his shopping cart. This repeats 10 times, after that client creates order, retrieves invoice information and pays for it. The application consists of database queries and two harder jobs, which represents possible real world task. The first is CPU intensive task will be done during creation of order, where application generates PDF and also calculates 41st Fibonacci number to add more cpu load. The second big task, which represents waiting for 3rd party service is done during handling payment, which is implemented as 500 ms sleep.

In Table 3.1 are results of benchmark which will act as baseline. Both Monolith and Modulith were running in single instance with 0.5 CPU assigned and Microservices assigned 0.5 CPU per service. Even though both Monolith and Modulith are running as single process, there is already little overhead visible due to more complex internal structure and more database connection pools (separate database pool is created for every module).

The biggest bottleneck of the first scenario is the CPU-bound task. To scale monolith it would have to run in more instances, which is very resource intensive for large systems, and the aim of this thesis is to focus on more modern architectural styles, so it will not be discussed further. On the other hand, for Modulith systems it is much simpler due to its modular structure. The CPU-bound tasks are contained in the invoice package, which can be moved to a separate

Benchmark performance scenario					
Architecture	Resources	rps	avg	p(90)	p(95)
Monolith	0.5 CPU, 50 MB	6.8 req/s	1.43 s	1.2 s	14.4 s
Modulith	0.5 CPU, 50 MB	7.8 req/s	1.25 s	1.59 s	9.18 s
Microservices	2.5 CPU, 250 MB	6.9 req/s	1.43 s	1.1 s	17.9 s

Table 3.1 Table containing benchmark results comparing Monolith, Modulith and Microservices. Microservices and much more CPU reserved, since it consist of 5 services (every singe one has 0.5 CPU assigned).

Benchmark performance scenario					
Architecture	Resources	rps	avg	p(90)	p(95)
Modulith unified	0.5 CPU	7.8 req/s	1.25 s	1.59 s	9.1 s
Modulith (1x invoice)	1.0 CPU	7.3 req/s	1.35 s	1.0 s	18.6 s
Microservices (1x invoice)	2.5 CPU	6.9 req/s	1.43 s	1.1 s	17.9 s
Modulith (2x invoice)	1.5 CPU	18.8 req/s	510 ms	605 ms	3.1 s
Microservices (2x invoice)	3.0 CPU	21.4 req/s	441 s	612 ms	3.9 s
Modulith (4x invoice)	2.5 CPU	44.2 req/s	204 ms	521 ms	1.2 s
Microservices (4x invoice)	4.0 CPU	58.3 req/s	165 ms	513 ms	1.0 s
Modulith (8x invoice)	4.5 CPU	71.6 req/s	133 ms	512 ms	0.8 s
Microservices (8x invoice)	6.0 CPU	66.4 req/s	142 ms	519 ms	0.9 s

Table 3.2 Table containing benchmark for Modulith and Microservices with module invoice moved into separate service running in multiple number of instances (indicated by number in parentheses).

lightweight service and scaled as desired.

In second measurement the invoice package was moved into separated service implementing HTTP server. HTTP client is than passed in main server to whoever module required the invoice interface. The same benchmark was run for one, two, four and eight instances of invoice service and results can be seen in Table 3.2 along with scaled Microservices example. There is no difference between Modulith and Modulith with single invoice service instance, but having the invoice service scaled to two instances, it has more than doubled the speed. Adding more instances scales linearly until the main bottleneck becomes IO bound task during payment. Microservices scale as well as Modulith, but having more overhead (more network inter-service communication) they are slightly slower.

3.6.2 Latency scenario

The second scenario consist just of the iterations part of previous scenario as defined on Figure 3.4, but instead of repeating 10 times as in first scenario, it will repeat 100 times. Benchmark will send totally 2 000 requests (100 iterations of scenarios, 100 iterations in scenario, 2 requests per iteration), where handling of requests will consist of multiple database queries, so application will be most of the time waiting for network or database.

Benchmark will be running 100 iterations consisting of 3 http requests to retrieve list of items, add random item to shopping cart and later remove it from shopping cart. For Monolith and Modulith it is just matter of inter-process communication and database queries. In case of Microservices, there is overhead in network inter-service communication. To handle either add or remove item, there are two network requests involved: first to retrieve item data to validate if the id of item is valid and second request to load data for all item ids contained inside shopping cart.

Results of running benchmark scenario are in Table 3.3. After running initial tests, there was unexpected behavior of Monolith, which has been much slower compared to Modulith (367 req/s

Benchmark latency scenario					
Architecture	Resources	rps	avg	p(90)	p(95)
Monolith	0.5 CPU, 50 MB	367 req/s	6 ms	51 ms	67 ms
Monolith	0.5 CPU, 100 MB	546 req/s	6 ms	60 ms	75 ms
Modulith	0.5 CPU, 50 MB	571 req/s	6 ms	68 ms	75 ms
Microservices	2.5 CPU, 250 MB	130 req/s	82 ms	107ms s	173 ms

Table 3.3 Table containing benchmark results comparing Monolith, Modulith and Microservices for second scenario.

vs. 571 req/s). During inspecting the runtime behavior via gathered OpenTelemetry metrics, it was found out, that application used up all the available memory and was not able to process more parallel requests. Assigning it 100 MB of memory fix the issue, although it surprisingly still did not supersede the Modulith in terms of performance. Microservices had the worst performance even though they had assigned much more resources, but the benchmark is actively pressing only two microservices: cart and item, and there is the overhead of network communication between them, which has to go through the whole network stack compared to inter-process communication in other two examples. The network communication added in average 24 ms to every inter-service communication, which caused inevitable reduction in performance (four times compared to Modulith) - this metric has been measured during running benchmark using OpenTelemtry tracing.

This huge performance difference in Microservices was much more than what was initially expected around 10 %. After looking into tracing and resource usage statistics, the main bottleneck was evidently database, where execution time of queries ranged from hundreds microseconds up to more than 5 seconds. This has been a big surprise, since Modulith application is making exactly the same database queries, but the issue wasn't there. There came some database optimization in play where depending on order of queries database was able to optimize it and in case of Modulith application the order was just better.

All benchmarks were run on a single machine, where it does not properly demonstrate performance drawback of distributed system as Microservices are. Even though there is network communication, the latencies are pretty low compared to actually running on multiple nodes, which would add even more latency. To demonstrate this on single machine, Linux Traffic Control has been used, which is very useful Linux utility that gives ability to configure the kernel packet scheduler to modify any network property [34]. After spawning the containers the *tc* command was used to set manually latency to all packets, which simulates real-world environment.

Table 3.3 contains benchmark results from running latency scenario with added extra latency to every running service. For modulith the extra latency is just on request coming from the client in both directions (inbound and outbound traffic). In case of microservices the extra latency is added for every service, so when latency is 2 ms, the inter-service communication actually has latency 4 ms, because two services are involved in communication. The first half of the table contains measurements of running applications with actual database, and again it shows very unexpected behavior, where Modulith with added just 8 ms latency is performing even worse than Microservices, which was not expected at all (Modulith should be always faster due faster inter-process communication). The issue again, as in previous benchmark, has originated in database now in favor of Microservices.

To mitigate the inequality caused by database, all the database queries has been replaced by sleep for 2 ms, which should simulated average database processing time and the results of running benchmark without database are present in seconds half of the Table 3.4. When looking on the results it is important to remind, that the benchmark is sending requests sequentially via 10 parallel clients. If it was just sending how many requests per seconds the application can handle the results would be basically the same for Modulith since the added latency would just project itself into total processing time of single request and the similar thing would happen

Benchmark latency scenario with database					
Architecture	Extra latency	Resources	rps	avg	p(90)
Modulith	0 ms	0.5 CPU, 100 MB	571 req/s	6 ms	68 ms
Modulith	2 ms	0.5 CPU, 100 MB	167 req/s	57 ms	83 ms
Modulith	4 ms	0.5 CPU, 100 MB	85 req/s	57 ms	164 ms
Modulith	8 ms	0.5 CPU, 100 MB	42 req/s	226 ms	330 ms
Microservices	0 ms	2.5 CPU, 250 MB	130 req/s	82 ms	107 ms
Microservices	2 ms	2.5 CPU, 250 MB	127 req/s	69 ms	107 ms
Microservices	4 ms	2.5 CPU, 250 MB	110 req/s	85 ms	126 ms
Microservices	8 ms	2.5 CPU, 250 MB	60 req/s	161 ms	232 ms
Benchmark latency scenario with database replaced by static 2 ms sleep					
Modulith	0 ms	0.5 CPU, 100 MB	668 req/s	14 ms	17 ms
Modulith	2 ms	0.5 CPU, 100 MB	566 req/s	17 ms	19 ms
Modulith	4 ms	0.5 CPU, 100 MB	457 req/s	22 ms	23 ms
Modulith	8 ms	0.5 CPU, 100 MB	318 req/s	31 ms	34 ms
Modulith	16 ms	0.5 CPU, 100 MB	200 req/s	50 ms	54 ms
Microservices	0 ms	2.5 CPU, 250 MB	498 req/s	14 ms	17 ms
Microservices	2 ms	2.5 CPU, 250 MB	289 req/s	30 ms	34 ms
Microservices	4 ms	2.5 CPU, 250 MB	202 req/s	46 ms	50 ms
Microservices	8 ms	2.5 CPU, 250 MB	126 req/s	78 ms	81 ms
Microservices	16 ms	2.5 CPU, 250 MB	69 req/s	143 ms	151 ms

Table 3.4 Table containing benchmark results comparing Monolith, Modulith and Microservices for second scenario.

for Microservices where it would project multiple times. But since the benchmark is running sequentially the added latency accumulates per every request thus slowing down the speed how the requests are being sent. From the results it is clear that latency negatively influences both architectures, but Microservices are more affected. Moduliths rps (requests per second) slows down by 30 % with added 4 ms latency and another 40 % with 8 ms latency, totalling 70 % slowdown, but with Microservices the situation is much worse. Added 4 ms latency for Microservices is slowing down rps by 60 % (this is twice more compared to Modulith) and with 8 ms latency another 26 % (notice how the another 4 ms latency is not nearly influencing as much as the first 4 ms, due to already big network overhead), totalling 86 %. The same negative effect has the latency on other metrics as well, where the average request takes for Modulith just 50 ms and for Microservices 143 ms, which is 2.86 times slower.

3.7 Summary

Three sample applications have been implemented, each using a different type of architecture. One used a monolithic architecture, the second a modular architecture and the last a microservices architecture. The different impact on the way the data has to be structured in the database and on the internal structure of the application depending on the architecture used was demonstrated. During benchmarking, the database caused unexpected application slowdown, leading to its removal and replacement with static sleep for the latency benchmark scenario to obtain more representative results.

Chapter 4
Methodology

If you are building an application from scratch, or just looking for a flexible architecture, I strongly believe that the Modulith is currently the best choice for the majority of projects. Based on the architecture summary from the first chapter Table 1.6, the Modulith shares the best qualities of both monolithic and microservices architectures. My experience from financial projects in chapter 2 argues against microservices as an unnecessarily complex architecture and the benchmarks in chapter 3 confirm the superior performance of Modulith and the same ability to scale with the hybrid Modulith approach. Based on all the work and results, I have created a methodology that can be used as a guide for architecture decisions when starting a new project:

1. **Modularization -** take the whole business domain and start dividing it into smaller parts until you reach single business responsibility per part. For each part properly define the scope and boundaries. Modeling around business domain has proven for Microservices to be the best approach[15], and it can easily be applied for Moduliths as well. This step usually impossible to get right the first time, so it is a more iterative process.

2. Create module for each defined part from previous step and expose its functionality over interface or messaging system.

3. Start composing modules to build the desired business logic resulting in Modulith architecture. The composition is done either by depending on exposed modules' interface or via external messaging system to keep the coupling as low as possible.

4. Prepare the application to scale as a unit (see Figure 4.1c). Running it in multiple instances is required to increase performance through horizontal scaling or to increase availability. For a successful transition, the application must be stateless, and at this point there may be issues with synchronisation and parallel access, so be sure to test it thoroughly under load.

5. If deployment is slow or deployment process too complex, deploy modules individually (more about deployment at bottom of section 1.2).

6. If performance is an issue, utilize Hybrid Modulith architecture to scale individual modules independently. This should be implemented with caution, as it may not even be needed as long as the scalability achieved by the previous steps is sufficient. See section 4.3 for more details.

7. Enforce module boundaries throughout the application lifecycle and adapt them as business needs change. Failure to enforce boundaries leads to increasing module size and complexity, which inevitably slows the pace of development.

8. If there is a good reason, Modulith can be easily converted into Microservices, see section 4.4.
   - Once the need for Microservices subside, it can be converted back to Modulith, see section 4.5

▶ Note 4.1. In the case of existing monolithic applications, they can be iteratively converted to a Modulith architecture rather than completely rewritten as with microservices, see section 4.2 for more details.

4.1 Why Modulith over Microservices

If you are building an application from the ground up, I strongly believe the Modulith is currently the best choice for majority of projects. See Table 1.6 with architecture summary in the first chapter, based on which the Modulith has the best qualities from both architectures Monolith and Microservices.

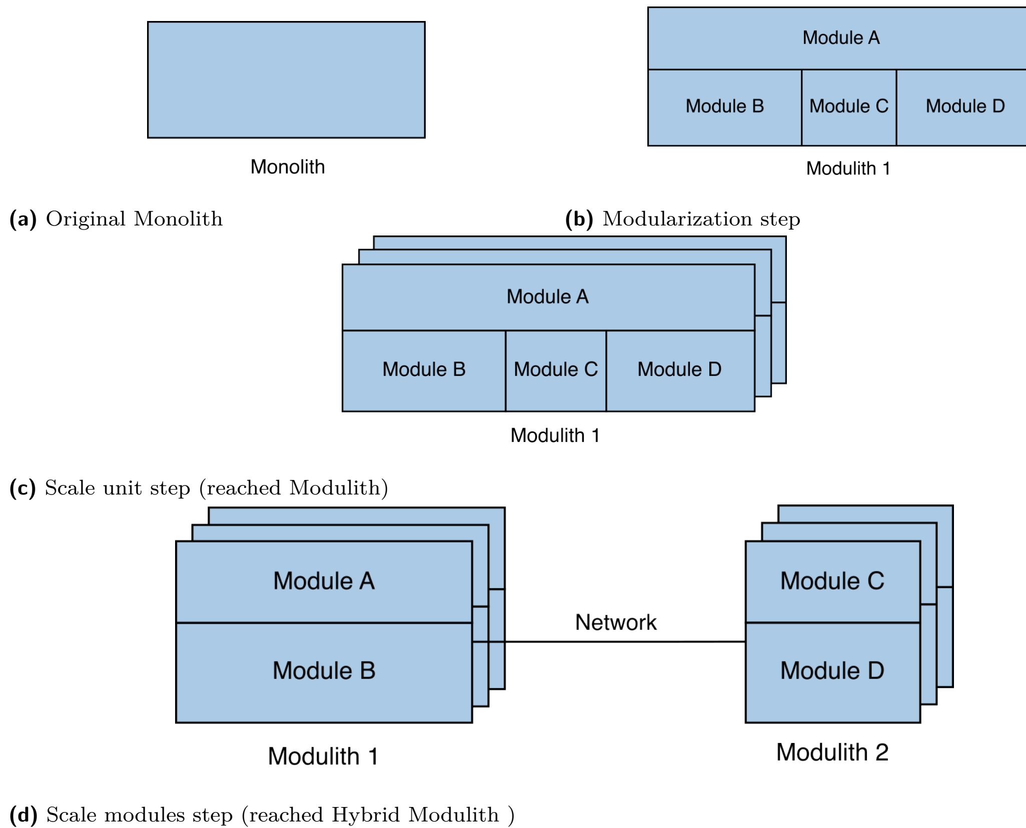
After all, why does everyone choose microservices in the first place? Because of performance? At the start of the project, there are some assumptions about the performance requirements, but they are still just "assumptions" and the actual performance issues are in most cases caused by the limitations of the actual implementation. In previous chapter, section 3.6, by benchmarking example implementation I have proven Modulith architecture to be superior in terms of performance, and Hybrid Modulith being superior in terms of scalability and resource usage (see Table 3.2). Also, the performance requirements can change dramatically if the business goes in a different direction than expected at the beginning and Modulith is more flexible to changes.

So, the reason could be the independence of microservices? The Modulith offers the same quality with modules. Only the microservices are more dramatically enforcing smaller scope, which can be leveraged for modules in the same way. What about testability? Again, modules can be tested in the same independent way as microservices, and even further using techniques known from monoliths such as end-to-end testing, without all the shenanigans associated with microservices and orchestrations. Maintainability? Modulith has a much simpler deployment strategy, less complex environment and is cheaper to host. Sustainability? The amount of work required to just merge two microservices into one or change API of single service is very high, since you can't even effectively find all the dependants and update them all accordingly. On the other hand, since Modulith is just a single codebase, the relationships are there and visible just by using standard and proven tools, e.g. IDE with 'find usage' feature. Also, any deep refactoring can be done in Modulith in a much easier way than in Microservice. The biggest negative aspect for small/medium projects lies in the ability of developers to properly define the scope and boundary of modules and more importantly to maintain it over time, as the individual microservices have a network boundary which is hard to cross, but with Modulith it is much easier as it is a single repository with inter-process communication, but this can be largely avoided by setting up automated tools which will enforce the boundaries at code level directly by monitoring cross-module dependencies.

4.2 Monolith to Modulith

Many software companies that have been around for a few years and did not jump into the microservices hype right away have some legacy monolithic systems that are costly to maintain, but they cannot be converted to microservices because it would require rewriting the whole system from scratch, and no one in the company wants to make that kind of big investment that would basically not bring any new functionality.

Instead of this radical approach of rewriting the whole system, which will basically cost the company twice the money, because rewriting the system means creating a new project with all the steps of development: analysis, design, etc. and long period of bug fixing. I would rather suggest



(a) Original Monolith

(b) Modularization step

(c) Scale unit step (reached Modulith)

(d) Scale modules step (reached Hybrid Modulith)

■ **Figure 4.1** Conversion steps from Monolith to Hybrid Modulith

an iterative approach of transforming the existing monolith into a modulith by concentrating on solving the specific problem at hand, rather than trying to solve everything at once by rewriting the application. The method is shown on Figure 4.1 and consists of the following steps:

1. **Modularization** step imposes the restructuring of the code and the modularisation of the whole application, but without worrying about the problems associated with a distributed system (network reliability, latency, complexity). Proper focus can be invested in defining modules, their responsibilities and boundaries. Depending on the project, the scope of modules can vary from complex modules to individual responsibilities per module.

2. **Scale unit** step is to prepare the application to scale by running multiple instances. Monolithic systems are usually stateful or require some form of synchronisation for some of their functionality. In this step, all these potential or actual barriers to running multiple instances need to be resolved or replaced with solutions that can scale. Now the application can achieve high availability and increased performance by scaling the entire instance. The architecture has now reached the definition of modularity.

3. **Scale modules** step takes scalability even further by allowing individual modules or groups of modules to be scaled, rather than the whole application as a unit. By completing the step, the architecture reaches the state defined in this thesis as *Hybrid Modulith*. See 4.3 for more details.

4.3 Modulith to Hybrid Modulith

When scaling as a single unit is no longer sufficient, the Modulith can be converted to a Hybrid Modulith, which allows scaling with more granular control. Be very cautions about using this approach since it introduces significant negative effects:

- Application becomes a distributed system - network instability, higher latency, complexity, complicated debugging and logging.
- More complicated deployment - some form of versioning is required to ensure compatibility between individual modules.
- More code and tools - instead of just having interfaces and implementing them in separate modules, now they also need to be implemented to support network calls, so some client/server code needs to be implemented or ideally generated by additional tools.

If you are sure to continue, follow the steps:

1. Identify the module that is causing the performance problem
2. Move the module into a separate service, along with all its dependent modules, to eliminate network calls as much as possible, effectively creating two Moduliths with a network communication between them (Hybrid Modulith, see Figure 4.1d).
3. Scale both Moduliths independently according to load
4. Repeat the division into more Moduliths until sufficient performance is achieved. Note that using this step to the extreme may convert the architecture to microservices, which may be better suited in that case, see subsection 4.4 for more details.

4.4 Modulith to Microservice

Over time, the modular architecture may no longer meet the needs of the project. Examples of reasons: more fine-grained granularity of the application is required, teams need more independence, lack of technology agnosticism, or a more distributed solution is desired to achieve higher availability or scalability.

After properly weighing all the pros and cons, a decision can be made to migrate to a microservice architecture. The transformation should ideally take place in natural steps over time as requirements and system needs change. Modulith consists of individually scalable modules with defined boundaries. Moving to a microservice architecture is about refining these modules and transforming them into microservices by defining fine-grained boundaries. In cases where modules have small scopes, the transformation can be as simple as one module to one microservice. However, some modules may have originally had larger boundaries that do not comply with the Microservices ideology, which is usually driven by the principle of single business responsibility. So, before doing the migration, the modules needs to be broken down into smaller modules, which than can be mapped directly to individual microservices. The transformation step from Modulith to Microservices is shown in Figure 4.2.

4.5 Microservice to Modulith

Microservices architecture can become too complex to manage or simply too expensive to operate. An example of the former is described in detail in chapter 2. The solution can be to convert microservices to Modulith, which simplifies deployment, resulting in cheaper infrastructure, and increases developer productivity by removing the complexity of a distributed system.

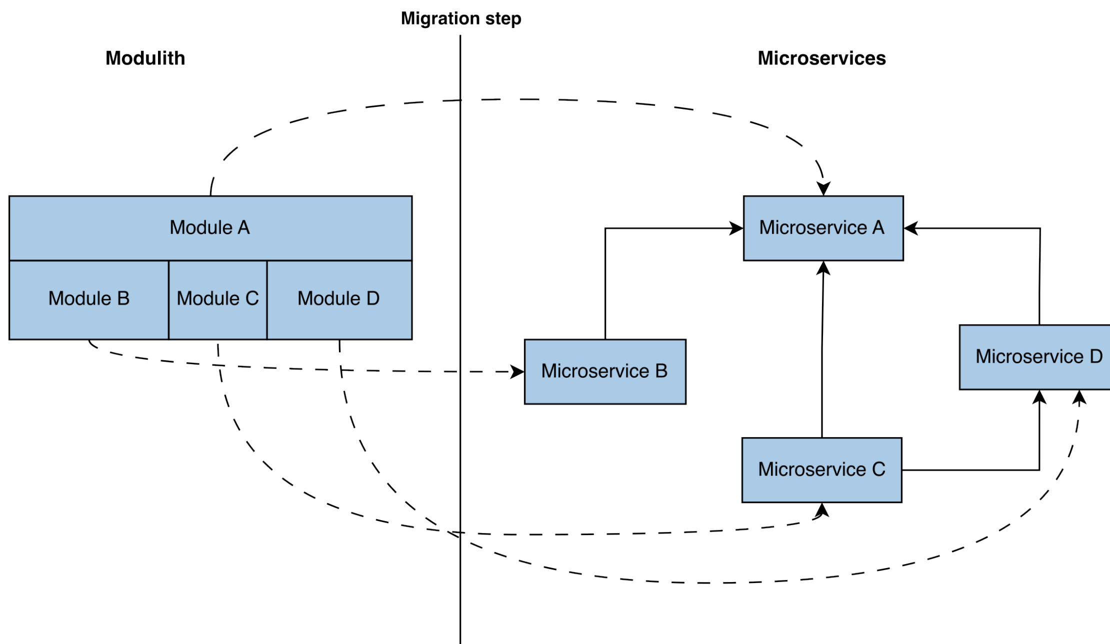


Figure 4.2 Conversion step from Modulith to Microservices.

The migration is easily done by converting microservices to modules as shown in Figure 4.3, as they are already fully encapsulated and have a defined API. Network communication can be completely replaced by inter-process communication, or if some modules require scalability for performance, they can be converted to *Hybrid Modulith* as specified in the *Scale Modules* step of section 4.2. Later, some modules may merge to take on more responsibility, but this depends on how module boundaries are set within the project.

4.6 Summary

The modular architecture methodology has been created as a step-by-step guide for new projects based on Modulith architecture. It defines how the business domain should be divided to modularise the application, how it should communicate internally to keep coupling low, and how it should evolve over time to increase performance and scalability. Each architecture step is detailed and illustrated with diagrams. The flexibility of Modulith even allows for a smooth transition back and forth between microservices and the said architecture.

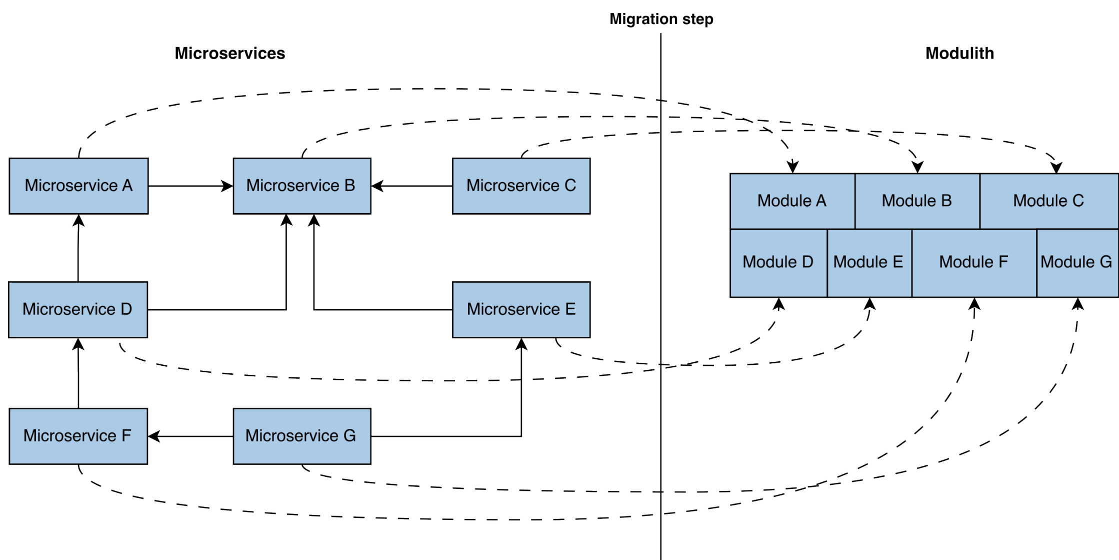


Figure 4.3 Conversion step from Microservices to Modulith.

Chapter 5
Conclusion

It is common for new systems to be built using overly complex solutions, encouraged by the distributed nature of microservice architecture, resulting in expensive projects. This thesis aims to highlight the negative aspects of microservices that are often overlooked and offer modern modular alternative approach. It compares the classical Monolithic architecture approach with Microservices and the modern modular architecture called Modulith. A detailed analysis of all three architectures has been performed, resulting in a matrix summarizing the architectures and their aspects. Modulith being the most flexible solution with the best qualities from both approaches. The ease of deployment of Modulith and the modularity and maintainability of Microservices.

I shared my personal experience as a backend developer working on a real project built from the ground up using a microservices architecture. The positive and negative aspects of the application cycle were discussed, including implementation, deployment and maintenance, while highlighting the potential for escalating complexity if microservices are not managed properly.

Proof of concept (PoC) has been created for each mentioned architecture type and discussed along with the internal application structure and database schema implications. Each PoC has been benchmarked for performance and latency, and the results have been analysed. The Modulith has been shown to outperform Microservices in terms of performance and resource usage. The Hybrid Modulith approach has also been shown to match the ability to scale as much as microservices.

The methodology has been developed using a modular approach based on the analysis, personal experience and results. It serves as a guide for making architecture decisions when starting a new project or migrating from other architectures. The foundation has been built on experience gained from Microservices, highlighting the importance of modularization for long-term maintenance and adaptability, rather than premature optimization to scale.

The objective of this work is to increase awareness of modern modular solutions that offer similar advantages to microservices, while avoiding the excessive use of microservices architecture in projects where the added complexity of a distributed system is unnecessary and only serves to increase development costs without any significant benefits.

Bibliography

1. SOFTWARE, CAST. *Software Maintainability: 75% of Your Budget Is Dedicated to Software Maintenance.* [WebPage]. 2023. Available also from: https://www.castsoftware.com/glossary/software-maintainability.
2. CONWAY, Melvin E. How Do Committees Invent? *Datamation.* 1968. Available also from: http://www.melconway.com/research/committees.html.
3. FOOTE, Brian; YODER, Joseph. Big Ball of Mud. 2003.
4. *Google Scholar* [WebPage]. 2023. Available also from: https://scholar.google.com/scholar?q=%22software+architecture%22&hl=en&as_sdt=0%2C5&as_vis=1&authuser=1&as_ylo=&as_yhi=2000.
5. *Google Scholar* [WebPage]. 2023. Available also from: https://scholar.google.com/scholar?q=%22software+architecture%22&hl=en&as_sdt=0%2C5&as_vis=1&authuser=1&as_ylo=2001&as_yhi=2023.
6. NEWMAN, Sam. Monolith to Microservices. In: O'Reilly, 2020, chap. The Monolith.
7. NEWMAN, Sam. Building Microservices. In: Second. O'Reilly, 2021, chap. The monolith.
8. NEIL FORD RebeccaParsons, Patrick Kua. Building Evolutionary Architectures by Neal Ford, Rebecca Parsons, Patrick Kua. In: First. O'Reilly, 2017, chap. Chapter4. architectural Coupling.
9. LAMPORT, Leslie. *Distribution.* Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87. 1987. Available also from: https://www.microsoft.com/en-us/research/publication/distribution/. Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87.
10. MANI, Ganesh. *Distributed Monoliths vs. Microservices: Which Are You Building?* [WebPage]. 2022. Available also from: https://scoutapm.com/blog/distributed-monoliths-vs-microservices#h_268859552881644344655113.
11. ERL, Thomas. Service-Oriented Architecture. In: Second. Mark Taub, 2017, chap. The monolith.
12. IBM. *What is service-oriented architecture (SOA)?* [WebPage]. 2023. Available also from: https://www.ibm.com/topics/soa.
13. BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for Agile Software Development.* Manifesto for Agile Software Development. 2001. Available also from: https://agilemanifesto.org/.

14. HAYWOOD, Dan. *How to build a modular monolith* [WebPage]. 2018. Available also from: https://conferences.oreilly.com/software-architecture/sa-eu-2018/public/schedule/detail/70817.html.
15. NEWMAN, Sam. Building Microservices. In: Second. O'Reilly, 2021, chap. What Are microservices?
16. NEWMAN, Sam. Monolith to Microservices. In: O'Reilly, 2020, chap. The Monolith.
17. FOOTE, Keith D. *A Brief History of Microservices* [WebPage]. 2021. Available also from: https://www.dataversity.net/a-brief-history-of-microservices/.
18. H, Jeremy. *4 Microservices Examples: Amazon, Netflix, Uber, and Etsy* [WebPage]. 2023. Available also from: https://blog.dreamfactory.com/microservices-examples/.
19. LTD., MarketsandMarkets™ Research Private. *Cloud Computing Market by Service Model (IaaS, PaaS, and SaaS), by Deployment Model, Organization Size, Vertical (BFSI, retail and consumer goods, telecommunications, IT and ITeS, and manufacturing) and Region - Global Forecast to 2027* [WebPage]. 2022. Available also from: https://www.marketsandmarkets.com/Market-Reports/cloud-computing-market-234.html.
20. KOLNY, Marcin. *Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%* [WebPage]. 2023. Available also from: https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90?utm_source=thenewstack&utm_medium=website&utm_content=inline-mention&utm_campaign=platform.
21. MÜLLER, Philip. *Under Deconstruction: The State of Shopify's Monolith* [WebPage]. 0009. Available also from: https://shopify.engineering/shopify-monolith.
22. SHOPIFY. *Shopify Engineering* [WebPage]. 2023. Available also from: https://shopify.engineering/.
23. LTD., MarketsandMarkets™ Research Private. *Netflix Announces Q4 2009 Financial Result* [WebPage]. 2010. Available also from: https://s22.q4cdn.com/959853165/files/doc_financials/quarterly_reports/2009/q4/NFLX_4Q09_Earnings_Release_012710.pdf.
24. GIRMONSKY, Alon. *Microservices Testability & Manageability* [WebPage]. 2020. Available also from: https://up9.com/microservices-testability-and-manageability.
25. ANDERSON, Charles. Docker [Software engineering]. *IEEE Software*. 2015, vol. 32, no. 3, pp. 102–c3. Available from DOI: 10.1109/MS.2015.62.
26. LLC, Google. *Kubernetes* [WebPage]. 2023. Available also from: https://kubernetes.io/.
27. HASHICORP. *Nomad by HashiCorp* [WebPage]. 2023. Available also from: https://www.nomadproject.io/.
28. JOSHI, Unmesh. Patterns of Distributed Systems. In: 1st. Addison-Wesley Professional, 2023, chap. Two Phase Commit.
29. RICHARDSON, Chris. *Pettern: Saga* [WebPage]. [N.d.]. Available also from: https://microservices.io/patterns/data/saga.html.
30. HASHICORP. *Consul by HashiCorp* [WebPage]. 2023. Available also from: https://www.consul.io/.
31. HASHICORP. *Consul by HashiCorp* [WebPage]. 2023. Available also from: https://www.vaultproject.io/.
32. *gorilla/mux* [WebPage]. 2023. Available also from: https://github.com/gorilla/mux.
33. PROJECT, Uptrace. *Bun: SQL client for Golang* [WebPage]. 2023. Available also from: https://bun.uptrace.dev.
34. HUBERT, Bert. *Linux manual page* [WebPage]. 2023. Available also from: https://man7.org/linux/man-pages/man8/tc.8.html.

Attached media contents

```
README.md .......................... brief description of the content of the media
golang ............................. directory of Golang implementation and benchmarks
  benchmark ........................ directory containing benchmark scenarios and results
    diff.sh ........................ helper to extract data from benchmark results
    diff_latency.sh ................ helper to extract data from benchmark results
    run.sh ......................... script to run benchmark for performance scenario
    latency_run.sh ................. script to run benchmark for latency scenario
    src ............................ JavaScript scenario definition for K6 tool
  docker ........................... docker-compose definitions for all variations of applications
    microservices
    microservices-no-db
    modulith
    modulith-no-db
    monolith
  server-microservices ............. microservices app implementation
  server-microservices-no-db microservices app implementation (DB replaced by sleep)
  server-modulith .................. modulith app implementation
  server-modulith-no-db ........ modulith app implementation (DB replaced by sleep)
  server-monolith .................. monolith app implementation
thesis ............................. source form of thesis in format LaTeX
thesis.pdf ......................... thesis in format pdf
```