Tuuli Sarantola

Migrating a Modern Web Application to the Cloud

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 6.7.2020

Thesis supervisor and advisor:

Prof. Petri Vuorimaa

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Tuuli Sarantola

Title: Migrating a Modern Web Application to the Cloud

Date: 6.7.2020 Language: English Number of pages: 8+56

Master's Programme in Computer, Communication and Information Sciences
Major: Computer Science Code: SCI3042

Supervisor and advisor: Prof. Petri Vuorimaa

Web technologies have been evolving fast since the adaptation of the modern web, and new tools and frameworks keep on coming into existence. With rapidly evolving technologies, the risk of accumulating technical debt becomes more common and must be taken into account at various stages of a software project. At the same time, new ways of working are generalising throughout development teams to ensure the quality of these projects.

The purpose of this thesis is to explore modern web technologies, methodologies, as well as different categories of technical debt that these might bring. It also takes a look into the public cloud and its most common cloud service providers and their services. Based on these, it proposes a strategy for bringing an existing web application onto a cloud environment.

Based on the case study project and literature evaluation, it is concluded that any application undertaking any larger work would benefit from having its technical debt at a manageable level and codebase in a good shape. It is also seen that when dealing with modern technologies, said technical debt might accumulate more rapidly than with projects implemented with more established technologies. When it comes to hosting web applications on the cloud, it is concluded that while a platform migration might bring some benefits in itself, a larger restructuring and careful redesigning work might be called for in order to fully reap the benefits of the public cloud.

Keywords: web, modern technologies, technical debt, public cloud, meteor, migration, modernisation

AALTO-YLIOPISTO PERUSTIETEIDEN KORKEAKOULU	DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Tuuli Sarantola

Työn nimi: Modernin Web-Sovelluksen Migraatio Pilviympäristöön

Päivämäärä: 6.7.2020	Kieli: Englanti	Sivumäärä: 8+56

Master's Programme in Computer, Communication and Information Sciences

Professuuri: Tietotekniikka	Koodi: SCI3042

Valvoja ja ohjaaja: Prof. Petri Vuorimaa

Web-teknologiat ovat kehittyneet nopeasti modernin nettiympäristön myötä, ja uusia työkaluja sekä sovellusalustoja syntyy jatkuvasti. Nopeasti kehittyvien teknologioiden kanssa työskennellessä teknisen velan kerryttämisen riski yleistyy ja sitä pitää tarkkailla monessa eri sovelluksen elinkaaren osassa.

Tämän työn tarkoituksena on tutkia moderneja web-teknologioita sekä metodologioita ja niiden mukana tulevan teknisen velan eri kategorioita. Työ tutkii myös julkipilveä alustana sekä tämän yleisimpiä palveluntarjoajia ja heidän tarjoamiaan palveluita. Näiden perustella ehdotetaan strategiaa, jolla modernin web-sovelluksen saa tuotua pilviympäristöön.

Tapaustutkimuksen ja kirjallisuuskatsauksen perusteella todetaan, että mikä tahansa sovellus, jolla on edessä isompi työkokonaisuus, hyötyy siitä, että sen kerryttämä tekninen velka on hallittavalla tasolla sekä koodikanta hyvässä kunnossa. Nähdään myös, että työskennellessä modernien teknologioiden kanssa kyseinen tekninen velka saattaa syntyä nopeammin kuin projekteissa, joissa on käytetty pidempään olemassa olleita teknologioita. Web-sovellusten tarjoamisesta pilviympäristöissä voidaan todeta, että vaikka migratointityö voi tuoda itsessään joitain hyötyjä mukanaan, voi isompi uudelleenjärjestämis- ja suunnittelutyö olla paikallaan, jotta voidaan saada kaikki julkipilven hyödyt irti.

Avainsanat: web, modernit teknologiat, tekninen velka, julkinen pilvi, meteor, migraatio, modernisointi

Preface

I'd like to thank my supervisor and advisor Petri Vuorimaa for his guidance during this thesis.

I'd also like to thank aTalent Recruiting, for giving me the opportunity to explore the cloud, as well as Nordcloud, for allowing me to continue on that journey.

A lot of time has gone by since I first started studying at Aalto, but every year brought something new. Thanks to all of the guilds, associations, and committees that I found along the way, it's been a blast. Also a special thanks to Joutomiehet, for making sure I wouldn't graduate too quickly.

Thanks to all of my family and friends for being there for me and helping me finish this. It took some encouragement, threats, weird ultimatums, wine, promises, and a quarantine for it to happen, but hey, here it is!

Otaniemi, 6.7.2020

Tuuli Sarantola

Contents

Abstract ii

Abstract (in Finnish) iii

Preface iv

Contents v

Abbreviations viii

1 Introduction 1
1.1 Research Questions 1
1.2 Research Methods 2
1.3 Limitations 2
1.4 Structure 2

2 Web Applications 4
2.1 Evolution of Web Applications 4
2.2 Frameworks 4
2.3 Architecture 5
2.3.1 Monolithic Architecture 5
2.3.2 Service-Oriented Architecture 6
2.3.3 Microservices Architecture 7
2.4 Design Approaches 7
2.4.1 Static Web Pages 7
2.4.2 Single-Page Applications 8
2.4.3 Multi-Page Applications 9
2.4.4 Progressive Web Applications 9

3 Modern Technologies 10
3.1 Modern Web Development Practices 10
3.1.1 Non-Functional Requirements 10
3.1.2 Test-Driven Development 11
3.1.3 Continuous Integration and Deployment 11
3.2 Modern Technology Properties 12
3.2.1 Risks of Emerging Technologies 12
3.3 The Meteor Framework 13
3.3.1 Development Speed 14
3.3.2 Usage and Popularity 15
3.3.3 Design Pattern 15
3.3.4 Data Handling 17
3.3.5 File Structure 18
3.3.6 Architecture 19
3.3.7 Hosting Options and Tools 20

4 Technical Debt in Software Development 22
4.1 Software Development Life Cycle 22
4.2 Technical Debt 23
4.2.1 Code Quality Debt 23
4.2.2 Architectural Debt 24
4.2.3 Non-Functional Debt 24
4.3 Dealing with Technical Debt 25
4.3.1 Code Quality 25
4.3.2 Architecture Quality and Migration 27
4.4 Special Considerations in Modern Technologies 28

5 The Cloud Environment 29
5.1 Web Hosting Evolution 29
5.1.1 Hosting Services 29
5.1.2 Private Data Centers 29
5.2 Virtualisation and Containers 30
5.3 The Cloud 31
5.3.1 Private Cloud 31
5.3.2 Hybrid Cloud 31
5.3.3 Public Cloud 32
5.4 Cloud Services 32
5.5 Benefits and Drawbacks of CSPs 34

6 Case: Project WorkTracker 37
6.1 Application Description 37
6.1.1 Features 37
6.1.2 Views 38
6.1.3 User Roles and Permissions 40
6.2 Technical Description 41
6.2.1 Technology Stack 41
6.2.2 Development Practices 41
6.2.3 Architecture and Design 41
6.2.4 Hosting and Deployment 42
6.3 Technical Problems 42
6.4 Goals 43

7 Migrating the Application 45
7.1 Codebase Update 45
7.1.1 Goals 45
7.1.2 Implementation 45
7.1.3 Results 46
7.2 Database Migration 47
7.2.1 Goals 47
7.2.2 Implementation 47
7.2.3 Results 47

7.3 Hosting Service Migration 48
7.3.1 Goals 48
7.3.2 Implementation 48
7.3.3 Results 49
7.4 Architecture Transformation 49
7.4.1 Goals 49
7.4.2 Implementation 50
7.4.3 Results 50

8 Conclusions **52**
8.1 Prerequisites of Migration Projects 52
8.2 Modern Technology Considerations 52
8.3 Hosting on the Cloud 53

Abbreviations

SOA	Service-Oriented Architecture
ESB	Enterprise Service Bus
API	Application Programming Interface
SPA	Single-Page Application
MPA	Multi-Page Application
TDD	Test-Driven Development
CI	Continuous Integration
CD	Continuous Deployment
MDG	Meteor Development Group
MVC	Model View Controller
MVVM	Model View View-Model
NoSQL	Not Only SQL
DDP	Distributed Data Protocol
SDLC	Software Development Life Cycle
CSP	Cloud Service Provider
AWS	Amazon Web Services
GCP	Google Cloud Platform
GKE	Google Kubernetes Engine
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
FaaS	Function as a Service
SaaS	Software as a Service
DBaaS	Database as a Service
npm	Node Package Manager

1 Introduction

Web applications and technologies are constantly evolving, bringing with them new opportunities as well as risks in terms of application development. In the short age of web applications, the rise and fall of several technologies, as well as development practices, have already been witnessed, and the evolution of web science is still going strong, bringing innovations constantly.

Along with these rapidly evolving web technologies and increasing demand in these services, the public cloud has risen to try and answer some of the challenges web applications eventually face during their life cycle. Based on virtualisation technology, cloud services and their service providers have developed a wide range of specified services, all applicable for different tailored purposes, and meant to make the life of web application developers easier.

As web applications evolve, there might come a time when they surpass their original purpose and design, and must find a way to adapt to changing requirements. [1] This might mean changing anything from the technologies, tools, or services used by the application. Often the application design and architecture might become obsolete, and must be reassessed if the application is to evolve.

In order to keep an existing web application alive, a modernisation process might be undertaken. With the rising popularity of cloud services, different migration strategies are often proposed as a means of modernisation. What this thesis aims to discover is, is a migration to a cloud-based environment the way to go when dealing with evolving web applications? And is it a solution that might benefit most of the applications developed with modern web technologies?

1.1 Research Questions

The goal of this thesis is to see if there are any special considerations to take into account when planning a migration of a web application implemented in a modern technology onto a cloud platform. It seeks to explore the concept of modern web applications and practices, as well as cloud services, to see what benefits might be gained when combining the two. It also considers what risks and drawbacks might occur in such an undertaking.

In light of the aforementioned goals, this thesis revolves around the following research questions:

- What are the prerequisites of migrating a web application to the cloud?
- What needs to be taken into account when dealing with modern technologies?
- What benefits and drawbacks are there in hosting a web application on the cloud?

1.2 Research Methods

To fulfill the set goal and answer the research questions mentioned above, this thesis will start with a literature review exploring the definitions of web applications, modern technologies, technical debt, and cloud environments. It will continue by looking closely at a case study of a software project implemented in a modern technology, exploring its current settings, key technical problems, as well as what goals are hoped to be achieved with a modernisation and migration process. Based on the information gathered in the literature review, it proposes and implements a migration strategy to a cloud-based environment, reflecting on what this process might entail and what steps are required in order to successfully achieve the modernisation and migration of the case application.

1.3 Limitations

This thesis will not explore the advantages and disadvantages of modernising legacy applications written in any older technologies. It will also not go into detail about the advantages of hosting applications on local premises. This thesis will only explore the process and advantages and disadvantages of migrating modern applications to a cloud-based environment, and what future actions this might entail for the application in order to achieve its modernisation goals.

1.4 Structure

Chapter 1 gives an introduction to the motivation, goals, and research methods of this thesis.

Chapter 2 begins by presenting the ecosystem surrounding web applications, their history, and tools. It continues by describing common architecture choices as well as design solutions.

Chapter 3 examines modern technologies and their specifications, as well as any special features they may have in the context of this thesis. It also presents the framework Meteor and its technical implementation and design solutions.

Chapter 4 continues by explaining the concept of technical debt in software development. It divides technical debt into different categories, and presents various strategies for dealing with them. It goes on to provide insight on what considerations should be made when dealing with modern technologies.

Chapter 5 explores the evolution of web application hosting, as well as the development of the technologies required for cloud services to rise. It then presents the most common public cloud providers, their services, as well as analyses their possible benefits and drawbacks.

Chapter 6 explores a case study of a modern time tracking web application and its current state, as well as its implementation details and technical problems. It lays out the desired goals for the modernisation process proposed by this thesis.

Based on the research, chapter 7 presents a strategy for the application's migration onto a cloud-based environment. It explores these steps in detail, as well as analyses their achieved benefits in respect to the application's migration goals.

Lastly, chapter 8 reflects on the modernisation process and summarises what can be generalised from it to answer the research questions.

2 Web Applications

This chapter presents an overview of web applications, their history, as well as different approaches to their development. A few common architectural styles are presented, as well as the most common design methods and tools used in web development.

2.1 Evolution of Web Applications

At its beginning, the web consisted mostly of text documents, static pages that might have contained hyperlinks to other documents. [2] These types of pages cover most of all early traditional web pages, whose sole purpose was to display information on a page and not much more.

The introduction of client-side web technologies, such as Flash and JavaScript, made way for the possibility of user interaction on web pages. This led to the birth of more complex web applications, where instead of just serving data, the application could respond to user input. Today, web applications can do much more than just display information to the user, and they are used for a wide and complex array of services, such as online stores, or personalised content sites.

Nowadays, the most popular client-side technology remains by far JavaScript. Multiple new libraries and frameworks written in JavaScript keep emerging frequently, expanding its original functionality and making it easier for developers to create complex web applications.

2.2 Frameworks

As the use of web applications started growing exponentially, tools such as libraries and frameworks, later referred to as frameworks in this thesis, were introduced to facilitate faster development cycles. These web frameworks, or web application frameworks, often come with a lot of advantages, making them popular among developers. Some advantages that have led developers to rely heavily on frameworks for web application development include having the use of readily available and standardised libraries, enhanced security through an active user base, and a clearer structure of code due to framework design and restraints.

Although they have many advantages, frameworks can also impact and application negatively, e.g. by complicating the codebase if used carelessly or in the wrong use cases, or if there are simply too many of them in a single project. Some frameworks have a lot of readily available code that might not be needed for certain applications, and they end up making the end product codebase messier and larger than it needs to be. Another thing that might suffer is code reusability between or in parts of projects, if the chosen tools vary considerably.

As all tools and technologies, frameworks have a limited life span, and not all frameworks are maintained long enough or updated often enough to meet their user's requirements. If a chosen framework's support is discontinued suddenly, it might lead to a massive project rewrite, or even contribute to a project's discontinuation. This is why when choosing technologies for a project, it is generally recommended to use well-established frameworks with a solid user base and maintainers, instead of opting for freshly released ones that might not yet be stable.

2.3 Architecture

There are different approaches to building web applications, which are represented by different architectural styles. A web application's architecture represents how the data is handled in a single application. Each architecture style has its own benefits and drawbacks, especially when it comes to the development, scaling, deployment, and maintenance of an application.

2.3.1 Monolithic Architecture

A monolithic approach to an application's architecture tends to bundle together all of an application's functionalities, as they are developed and deployed as a single unit. This is the most straightforward and simple form of architecture, and usually the starting choice of architecture for new applications.

This approach works well for relatively small or non-complex applications, but as the application size and complexity start to rise, the development and maintenance of a monolithic application become more complex. Adding just a single feature or making a change in the application becomes much slower, as the changes affect the whole application instead of just a single, separate functionality within. This leads to reduced speed for development, testing, as well as deployment, and increases the chance of introducing breaking changes in the functionality of the application. [3]

Another aspect to take into account with monolithic applications is the difficulty of scaling, which eventually becomes relevant as an application starts to grow in size or user base. As the application scales up or down as a single entity, it's impossible to target only the needed functionality for scaling. This leads to scaling up parts of the application which wouldn't need additional resources, but are given them anyway. Usually, these applications are scaled vertically, meaning more processing power or memory is added to the machine hosting the application. This type of scaling becomes expensive quite quickly and isn't a sustainable choice if continued at length. [3]

Monolithic applications are perhaps the oldest and most traditional form of application architecture and can be encountered in many older, legacy systems. These monoliths are usually the product of many years of work by different developers and include outdated code mixed with newer services, which result in increased complexity. This in part explains the general reluctance of updating legacy application technologies or making any changes to their functionality.

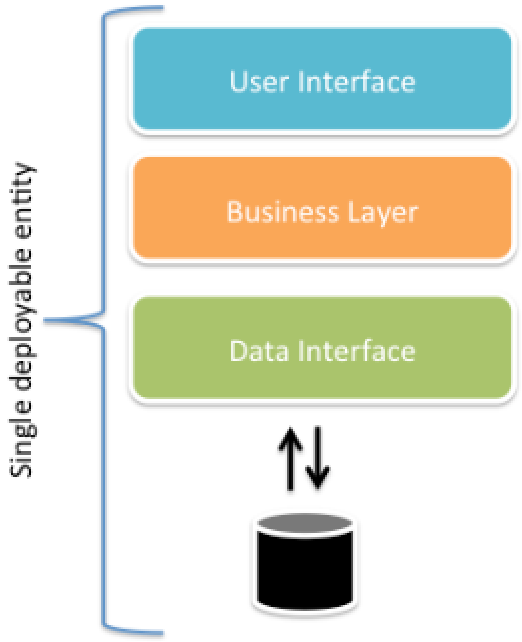


Figure 1: Monolithic architecture. [4]

2.3.2 Service-Oriented Architecture

A Service-Oriented Architecture (SOA) breaks down the application functionality into components, which provide services to each other via a communications protocol over a network. Each service is in charge of a single, smaller functionality, such as logging in a user. These services can then communicate, e.g., by passing data between correspondent services, or by orchestrating a bigger joined functionality. [5]

SOA components assume one of two main roles: service providers and service consumers. The consumer layer is the point where consumers, such as users or other services, interact with the SOA application. The provider layer consists of all the services defined within the SOA application. To communicate, SOA applications make use of an Enterprise Service Bus (ESB). The ESB is a middleware tool used to distribute work among the application components, connecting and activating the needed components of the application as needed. [5]

Although the SOA approach is less centralised than that of a monolithic application, some risk areas still exist as the application grows in complexity and size. As a sort of monolithic service of its own, the ESB can become a vulnerability. Since every service is communicating through it, any problems with any one service may cause the ESB to get clogged up with requests for that service, slowing down and affecting the entire application.

The drawbacks of scaling a monolithic application are not present in an SOA application, as the different services can be individually scaled up or down, depending on their usage, thus reducing the unnecessary allocation of computing power or memory. The deployment process, however, is still dependent on the whole application. Even though individual services may be developed independently, the deployment of a new version must be done throughout the whole application. [5]

2.3.3 Microservices Architecture

The microservices architecture style is an approach to developing an application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often a Hypertext Transfer Protocol (HTTP) resource Application Programming Interface (API). These services are focused on a single business capability and are independently deployable and scalable. This allows developers to focus on only specific parts of the application, without having to worry about affecting other services. They may even be developed in different technologies and maintained by different teams. [6]

The microservices architecture style is often considered a subset of SOA, both of them having many overlapping qualities and principles, such as centralising services around business capabilities and having a distributed system of services. [7] Although they have many similarities, a key difference is that the microservices architecture model focuses more on achieving agility and simplicity at the business level, avoiding the complexity of centralised ESBs. [1]

While both SOA and microservices architecture applications achieve agility for the development process of an application, the microservices model manages to bring that agility to the deployment and maintenance aspects as well. While SOA applications have to be deployed as a whole, a service of a microservices architecture application can be updated or shut down without having any impact on the other services. [5]

2.4 Design Approaches

Apart from architectural design decisions, there are other aspects to take into account when developing web applications. A few of these design approaches are explored below.

2.4.1 Static Web Pages

Static web pages date back to the beginning of web pages and are mostly used only to display information, not to be responsive to user input. These are used solely to show content to users, e.g., in the information of blog pages.

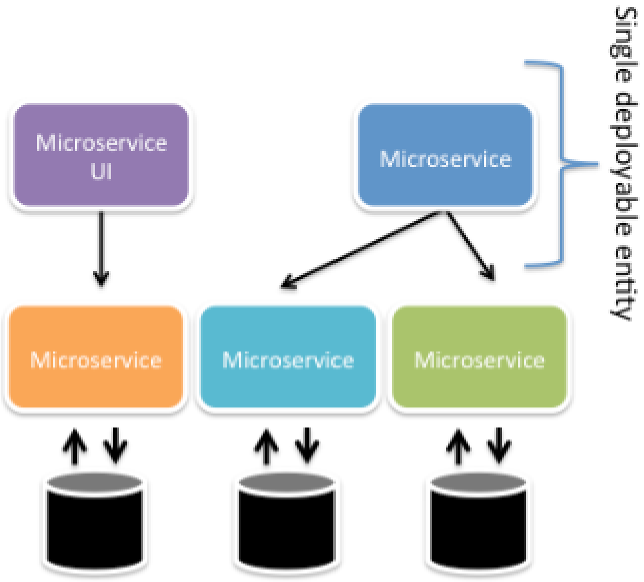


Figure 2: Microservices architecture. [4]

Static web pages as such don't contain any additional application logic, which makes them fast to load, but unresponsive to user input. In practice, static web pages are often mixed within a broader web application, where other parts might make use of some client-side logic to handle user input.

There exist a number of tools for generating static sites, which make the development experience closer to that of Single-Page Applications (SPAs) or Multi-Page Applications (MPAs), but result in a number of static web pages, ready to be deployed as such. Such tools include, e.g., documentation and blog generators.

2.4.2 Single-Page Applications

SPAs are applications that load application logic into the client-side and work inside the browser. Unlike MPAs, instead of sending whole web pages as responses to updated data or user input, the application can update just parts of itself, making the whole application seem to live on a single page, increasing interactivity and responsiveness greatly. [8]

The development of client-side JavaScript and its advanced frameworks, such as Angular or React, was a prerequisite to the SPA design approach. This approach makes the client-side heavier in terms of application logic than traditional MPA design did, but is usually considered a decent trade-off for the improved user experience of the application. [9]

Having all the content loaded only after the initial web page has its drawbacks, especially regarding Search Engine Optimisation (SEO) or slow internet connections. This leaves site crawlers and low-end internet users with an empty shell of a web application before any meaningful content is loaded. However, some technologies have been created to address these problems, such as server-side rendering, which prerenders some of the application content before sending it to the client. [10]

2.4.3 Multi-Page Applications

MPA design is the traditional approach to web applications. Every change, such as input from a user or displaying changed data, requires sending data to the server and rendering a new page, thus refreshing the client-side. MPAs are often quite large and complex, dealing with a lot of information, meaning that there is heavy traffic of information between the client and the server.

The introduction of Asynchronous JavaScript and XML (AJAX) led to a hybrid design of MPA and SPA. Instead of having to refresh the whole application on user input, AJAX made it possible to refresh only a part of the application in the style of SPAs. This approach makes the application more user-friendly, as page refreshes are less frequent and the interaction is more responsive. However, adding an AJAX interface makes an MPA more complex and difficult to develop. [9]

Nowadays, MPAs are still partly in use, although they tend to be part of a hybrid application. Many applications might also seem to be MPAs at first sight, but are actually SPAs making use of different routing libraries to simulate the flow of MPAs.

2.4.4 Progressive Web Applications

A slightly different approach to web applications is the progressive web application approach, which is meant to offer a native-like experience using web applications. The idea is to have some native functionalities, such as offline usage and push notifications, available from a traditional web application. [11]

Progressive web applications are meant to be used cross-platform, and as such reduce the amount of work developers would otherwise have to put into developing separate web and mobile applications. As the application requires no installation, it is a lightweight approach to mobile applications. [11]

Nowadays, the popularity of progressive web applications has somewhat diminished, as different tools have risen up to make web applications portable to mobile devices, and different frameworks offer the tools to deploy an application for multiple platforms at once.

3 Modern Technologies

This chapter explores modern web development practices, as well as modern technology properties and their entailed risks. It also explores in detail the modern web application framework Meteor, especially its architectural and design approaches, in order to give context to the practical section of this thesis.

3.1 Modern Web Development Practices

As technologies evolve and new ones emerge, so do different development practices. A lot of different practices have become quite common in the modern web development world, and quite often software development teams have some sort of modern practices in use, or at least in consideration.

3.1.1 Non-Functional Requirements

Non-functional requirements define system attributes that have no functionality in themselves, such as security, as opposed to functional requirements, which are the main functional components of an application. While not a practice in themselves, non-functional requirements have become quite central to any modern web development projects. As such, taking various non-functional requirements into consideration while developing applications can be considered a modern practice in itself. [12]

Scalability. The capability of a system to handle a growing amount of requests or work is known more commonly as scalability. While not necessarily relevant at the beginning of a software project, the concept of scalability should be taken into account while designing any system that might grow in demand or users. Also taking into account the possible decrease or fluctuation in user amounts is a part of scalability considerations.

Availability. The degree to which a system is functioning and accessible to its users is known as availability. With the rise of almost always available services and service providers, users have grown used to a higher availability rate than previously. This means striving for an almost zero downtime in a service.

Performance. The amount of work achieved by a system, or its speed in doing so, is known as performance. In a world of faster connections than before, users are expecting a certain level of performance in terms of response time or successful transactions.

Data Integrity. The accuracy and consistency assurance of data during its life cycle can be described with the term data integrity. As opposed to having potentially corrupt data, an application should make sure that the data it stores is correct, without any erroneous writes or transactions that might affect the data negatively. Ensuring that the data is altogether available also falls into the domain of data integrity.

Maintainability. The ease of making changes or adding new sections to the codebase is known as maintainability. It is closely related to code quality and the minimising of technical debt, but can also cover the general ease of fixing faulty parts, or just ensuring the longevity of a system.

Security. The protection of a system and its data is commonly known as security. Security can cover a lot of different aspects, such as physical and digital security, where theft or damage to a part of the system or its data should be prevented. Security can be taken into account both in infrastructure and code design, and is often something that should be regularly checked, and if needed, updated accordingly.

Backups and Disaster Recovery. While ensuring that the system and its data are available and their integrity unbreached, backups and disaster recovery plans should be taken into account in case of sudden unforeseen changes in, e.g., the hosting server. While holding no value in themselves, backups and disaster recovery plans contribute to other non-functional requirements, such as data integrity, security, and overall availability of the system.

3.1.2 Test-Driven Development

Test-Driven Development (TDD) is a software development practice where tests are written incrementally prior to code implementation. This forces the developer to focus on the functionality of the code before actually implementing it. [13]

As TDD forces the development team to spend a lot of time writing tests and implementing code to pass them, it concentrates the focus of the team on important core functionalities, and away from not so critical features. This mentality complements an agile way of thinking, where functionality can be incrementally added and deployed, instead of trying to implement a whole system at once.

Although TDD can hardly be considered a new development practice, it has gained some new popularity during the last few years. A lot of the benefits that ensue from TDD, such as higher code quality, a smaller amount of bugs, and a more consistent codebase [13], are held in high esteem among modern web developers. As such, TDD can often be integrated into a development team's consistent development practices.

3.1.3 Continuous Integration and Deployment

Continuous Integration (CI) is a software development practice where developers integrate their code early and often, to reduce the work and errors of making larger and fewer integrations. This approach allows for faster feedback about the newly integrated code and its effects on the whole program. [14]

Continuous Deployment (CD) is a practice that goes a step further than CI. On top of testing and integrating the codebase in a development environment, it builds executables and pushes them to increasingly production-like environments to make sure the software will work in production. Often the actual deployment itself is part of the process as well, and most of the operational tasks involved in publishing software are automated as well. Processes that leave the deployment out of these pipelines are referred to as continuous delivery. [15]

Continuous integration and deployment (CI/CD) in themselves bring no additional value to development teams and software projects unless comprehensive test suites have been built as part of the pipeline. Once these are in place, a lot of bugs and breaking changes in functionality can be caught before they ever reach a production environment. [16]

While both the concepts of CI and CD have been around for a while, they have been widely adopted since as a crucial part of modern software development. A lot of tools specific to CI/CD have emerged, and many of the more popular version revision tools come with readily available CI/CD pipeline tools. Such tools can nowadays also be found in all of the biggest cloud service providers as part of their automation pipelines.

3.2 Modern Technology Properties

Web technologies have been evolving fast, and new modern technologies, frameworks, and features seem to pop up constantly. When talking about the fast evolution of these technologies, usually one of two things is meant:

1. New frameworks, libraries, or technologies are born

2. Existing ones have a fast release cycle of new versions or features

When talking about modern technologies, we can generalise that they encompass both of these qualities: they are relatively young and have a fast release cycle of new versions. Once a technology has already been around for a while and its development pace has slowed down and stabilised, we can consider it an established technology. Once established, a technology loses some of the risks commonly attributed to modern technologies.

3.2.1 Risks of Emerging Technologies

Since new technologies are born faster than they can become established, a certain amount of these eventually doesn't make the cut into well-established technologies. Once established, a technology is considered more stable to work with and carries fewer risks to bigger software projects.

End of Life. One of the biggest risks of working with emerging technologies is a sudden end of life, as their future development and maintenance are discontinued. Of course, nothing guarantees that even technologies that have been around for a while and in heavy use will continue to be maintained forever, but as a technology gains popularity, the amount of willing and capable maintainers grows as well, making it less likely to be deprecated completely.

Deprecated Versions. Less severe than discontinuing a technology completely, a risk with modern technologies is quickly deprecating earlier versions of said technologies. While it is not a requirement that old versions of any technology be maintained forever, it is good to take into account when starting a project that will most likely take some time to finish. If the release cycle of the chosen framework is a lot less than its lifespan so far, the project might suffer delays due to constant necessary updates to the existing codebase to keep up with the newer versions.

A good example of risks related to keeping up with version updates is the release of the Angular 2 framework. What was supposed to be a version update to the original Angular framework, nowadays known as AngularJS, turned out to be incompatible with its earlier version. This led the version update to be released as an entirely new framework. AngularJS is now in a controlled end of life process. [17]

When it comes to making technology choices at the beginning of a new software project, often older and thus seemingly more reliable technologies are preferred to younger ones. However, as technologies keep on evolving fast, it is a good practice to keep an eye on more modern technologies, as smarter design and new ways of implementing things are born that might surpass older technologies. It is up to the decision-maker of every new project to weigh the benefits and risks of choosing a modern technology.

3.3 The Meteor Framework

The Meteor framework was created in 2012 by the Meteor Development Group (MDG). It is a framework meant to facilitate the whole of web application development, encompassing the frontend, backend, as well as the database. As such, it can be considered a full-stack web application framework. [18]

Meteor is written in JavaScript and is based on Node.js. One of its key advantages is to be able to use the same language, JavaScript in this case, in any part of the application. It can also use the same application code in the back- and frontend, making it highly reusable. Due to its project and code structure, it also means that an application built with the framework is often monolithic by design.

Even though Meteor was designed to handle the full stack of a web application, it was later integrated with more popular frontend frameworks due to community demand. Instead of using Meteor's native frontend templating system, Blaze, developers could now also choose to use more familiar frontend libraries, such as Angular or React, and make use of the Meteor framework as more of a backend solution.

By default, Meteor ships with MongoDB as its database, and communicates with it through a JSON interface. Although Meteor still recommends using MongoDB in its applications, the usage of other document-oriented databases was also made possible. However, the use of traditional relational databases is still not supported in Meteor.

3.3.1 Development Speed

Although Meteor was started in 2012, its first public release, or version 1.0, was published in 2014 [19]. It is currently at version 1.10, having undergone many breaking changes, rewrites, and guide updates in the last 6 years.

From Figure 3, we can see that the first 4 years of development were the busiest, with a frequency of up to 200 commits at times. This indicates a lot of ongoing changes from the early stages of the framework, as it was still finding its final format and structure, especially leading up to its first major release. If we compare this to Figure 4, we see that the busiest time period ends up around Meteor's release 1.1, but development continues actively until around version 1.3. After this, the development speed seems to stabilise.

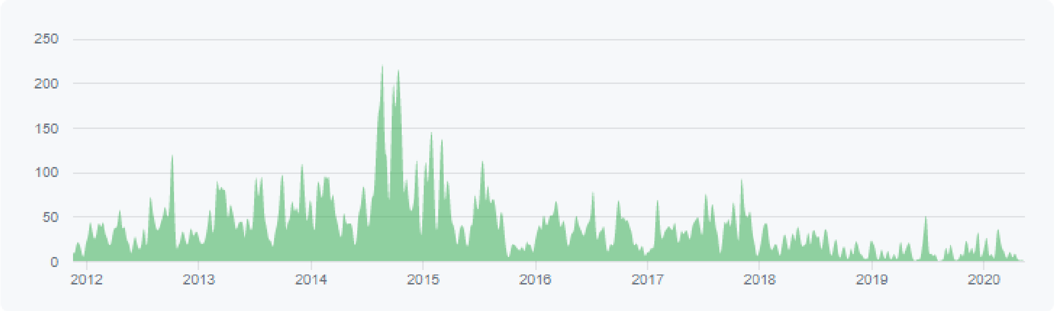


Figure 3: Meteor commit frequency. [20]

As we can see in Figure 4, Meteor's version release speed has stayed quite constant over the years, resulting in a major release around every 6 months or so. Only in the past year has the development speed seemed to decrease a little, resulting in only one major release per year. Since Meteor's developers MDG have been actively developing other projects as well, it seems only natural that the release curve would slow down eventually.

While the development speed itself might not give us much clue about the ongoing developments of the framework, another property that can be examined is the frequency of the breaking changes these releases have brought. During these 10 major releases, there have been such dramatic changes as a major version update to the underlying Node.js, a renewed file structure for applications, a new module import mechanism, as well as the possibility

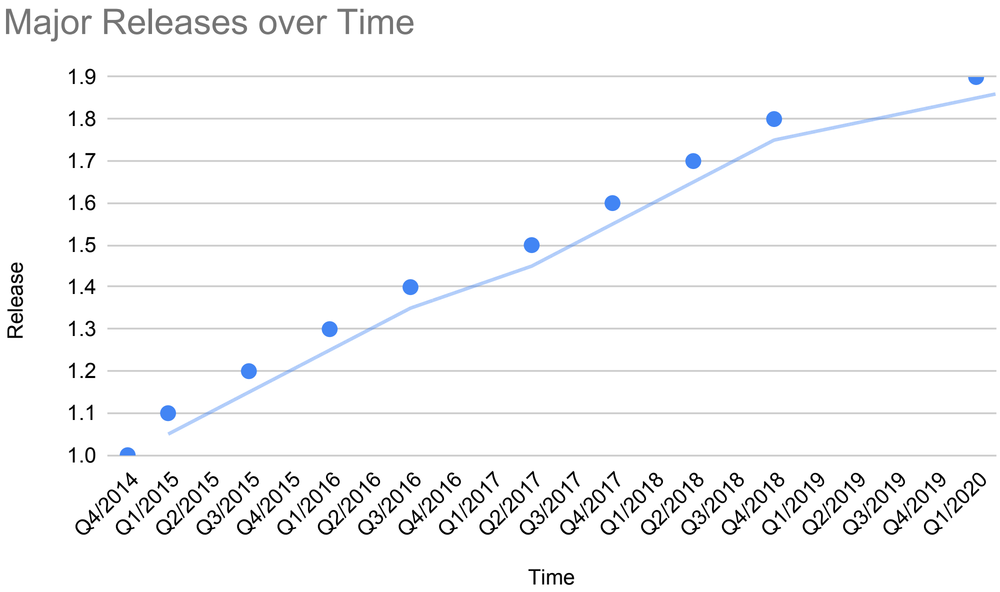


Figure 4: Meteor development speed.

to use npm packages instead of relying solely on Meteor's package system, Atmosphere. [21]

3.3.2 Usage and Popularity

As with all new and upcoming JavaScript frameworks, Meteor started with a few select users, but found a growing popularity after a few releases. There is an active community and forum of developers dedicated to using the framework, and occasionally contributing to it. This led to its growing popularity especially in the early stages of the framework.

Since Meteor allows for the use of any frontend framework, it is often compared to solely backend frameworks. As applications made with Meteor started to expand and grow, more issues with production environment related problems started to rise. This led to a switch in the choice of backend frameworks in applications originally started with Meteor. As we can see in Figure 5, community interest has had a steady decline over the last couple of years. When we compare this to, e.g., the Express framework, we see that as Meteor's popularity declines, the newer Express framework's popularity seems to grow. [22] Whether this is due to the appeal of a more modern framework, or just differences in technical implementation, is hard to say.

3.3.3 Design Pattern

Meteor applications are built on what they call a Model View View-Model (MVVM) architecture pattern. MVVM expands the concept of the nested

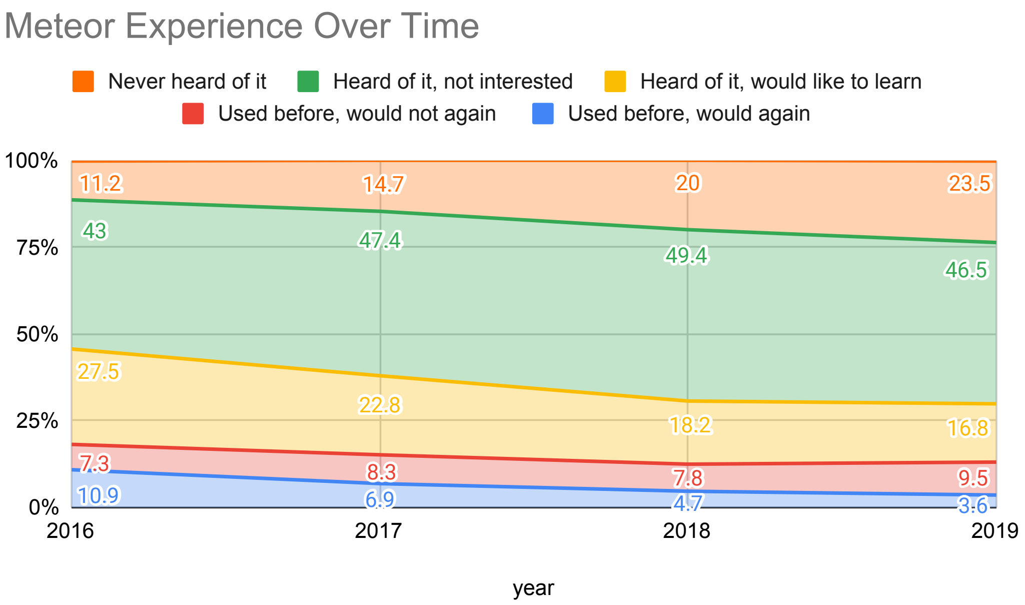


Figure 5: Meteor framework user experience over time. [22]

Model View Controller (MVC) pattern, where the view of the server-side application is used as the model of the client-side application, as seen in Figure 6. Meteor takes this a step further, as it brings the nested MVC concept to the frontend of the application, resulting in multiple separate MVCs.

The Model. Meteor has a model of cached and synchronised data, which is the same on the client and the server. When a change happens in the client data model, the change is first cached locally, after which it tries to synchronise with the server. The client also listens for changes coming from the server, which allows it to store a local copy of the data model. In this pattern, the results of any changes can be passed to the screen quickly, without having to wait for a server response. [18]

The View. The Meteor client renders HTML through the use of templates, or view data bindings. These are essentially a shared piece of data that will be displayed differently should the data change. Meteor uses Handlebars template expressions in its HTML. [18]

The View-Model. Meteor's client-side code is responsible for tracking changes to the model and presenting them in a way that the view can pick them up. The view-model is also responsible for listening to changes coming from the view, such as changes to a template value. This means that the client controller has its model (i.e., the data from the server) and the view has its model (i.e., a template) and both are responsible for knowing what to do with their respective models. [18]

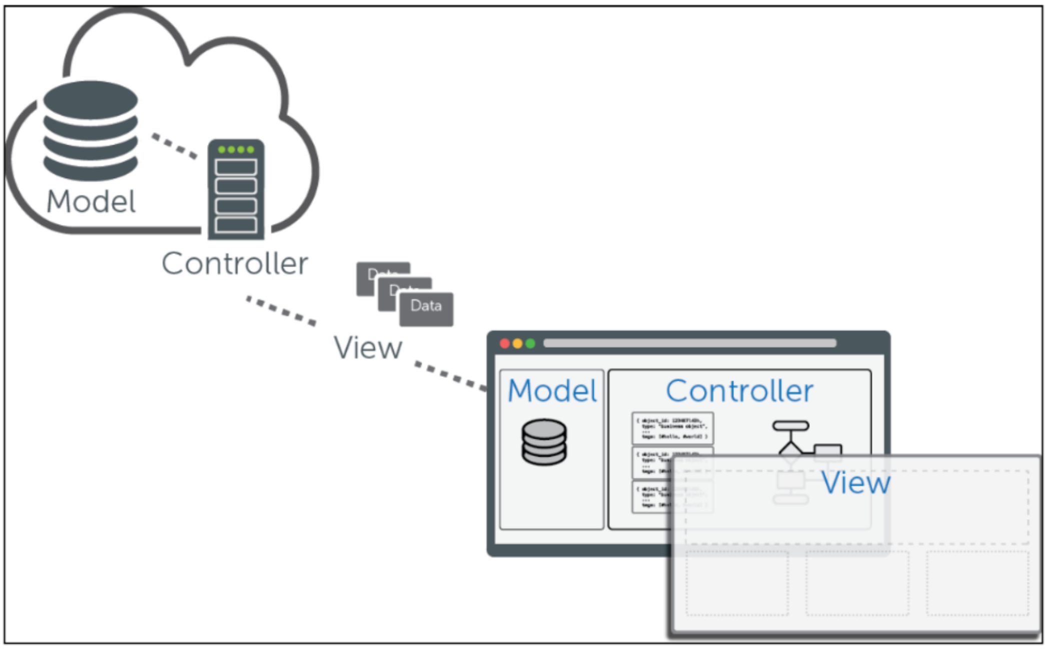


Figure 6: Nested Model View Controller pattern. [18]

3.3.4 Data Handling

The default database included in Meteor is the document-oriented NoSQL database MongoDB, and the data stored is referred to as collections. As mentioned in the design pattern section, Meteor stores a local version of the application models on the client. To do this, it uses a lightweight version of MongoDB called MiniMongo, stored on the client.

To listen for changes in its collections, Meteor uses a publish/subscribe model. When changes are made, a change event is published, while callback functions are subscribed to these published events. All code in such callback functions is activated when the specific event they are subscribed to is published. The developer is in charge of controlling when and where data is published or subscribed to, thus controlling the extent to which data is available to end-users.

Some of Meteor's built-in functions and variables listen automatically to these change events, making the whole application update in real-time, instead of polling for changes. This is the basis for Meteor's reactivity principle, where real-time data is centrally updated to all application clients, thus preventing different states of the same application to exist at the same time.

Meteor uses its Distributed Data Protocol (DDP) to communicate data changes throughout the application. This protocol keeps the databases in sync between the central database on the server, and every connected client. [19]

3.3.5 File Structure

As Meteor is a full-stack framework, any application made with it contains code that runs on the client as well as on the server. On top of this, they contain common code that can be run on both. However, all code isn't meant to be run on both the client and the server. For this purpose, Meteor proposes an application file structure model to control when and where code can be accessed, depicted in Figure 7. Following this structure, the developer can control what code is used and eventually bundled into the application itself.

As of version 1.3, Meteor has had full support for ES2015 modules, which have quickly become the industry standard. This module system supports making variables available outside a file using the `export` keyword, as well as using them somewhere else by using the `import` keyword respectively. In Meteor, files that are placed in the imports-folder must follow the ES2015 module structure to be made available to the rest of the application. [23]

Since prior to version 1.3 ES2015 module imports weren't supported, code was made generally available throughout the application. In turn, there weren't any best practices when it came to project file structure, and it wasn't exactly strict. Still, Meteor introduced some rules that are still present in later versions of the framework. These include a few specially named folders that Meteor treats differently. Any files outside of these folders are generally available throughout the application, both on the client and server sides. [23]

imports. Any file inside of this directory won't be loaded anywhere unless specifically imported into another file elsewhere in the application.

node_modules. As with the imports-folder, any Node.js packages installed into node_modules won't be loaded into the application unless imported.

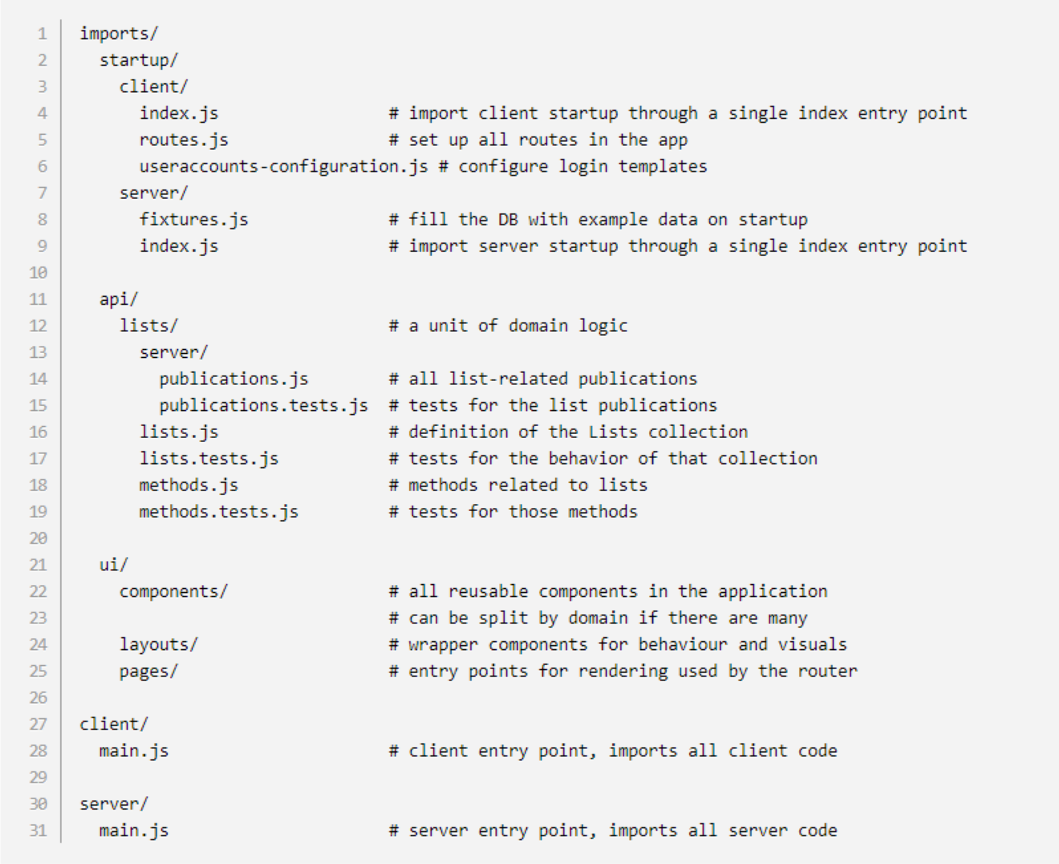
client. Any files within this directory will only be loaded onto the client-side of the application, making them unavailable to the server.

server. As with the client-directory, any files within this directory will only be loaded onto the server-side of the application, making them unavailable to the client. Any sensitive code that the developer wants to keep out of reach of the client should be placed in this directory.

public. Any files within the public-directory are served as they are to the client. For example, favicons or other static assets should be stored in the public-folder.

private. Any files within the private-directory are only accessible to the server-side, and only through Meteor's special Assets API. This can be used for private data files, or files containing sensitive information.

tests. This directory is meant for test files, and any files within it won't be loaded anywhere in the application. Instead, Meteor's built-in test tools make use of this directory to run any tests that might reside there.



```
imports/
  startup/
    client/
      index.js                 # import client startup through a single index entry point
      routes.js                # set up all routes in the app
      useraccounts-configuration.js # configure login templates
    server/
      fixtures.js              # fill the DB with example data on startup
      index.js                 # import server startup through a single index entry point

  api/
    lists/                     # a unit of domain logic
      server/
        publications.js        # all list-related publications
        publications.tests.js  # tests for the list publications
      lists.js                 # definition of the Lists collection
      lists.tests.js           # tests for the behavior of that collection
      methods.js               # methods related to lists
      methods.tests.js         # tests for those methods

  ui/
    components/                # all reusable components in the application
                               # can be split by domain if there are many
    layouts/                   # wrapper components for behaviour and visuals
    pages/                     # entry points for rendering used by the router

client/
  main.js                      # client entry point, imports all client code

server/
  main.js                      # server entry point, imports all server code
```

Figure 7: Example application structure. [23]

3.3.6 Architecture

Due to the data model, design, and application structure of the Meteor framework, applications often end up having a lot of shared code throughout their files and different parts of the application. This leads to ambiguous code, of which some is executed on the client, some on the server, and some on both. Since the framework is especially meant to make it easy for new developers to start making new applications, it can end up blurring the traditional lines of client and server code, and making it harder to distinguish which parts of the applications belong to which.

As mentioned earlier, this approach to application development leads to a monolithic architecture in applications developed with Meteor. Since code is meant to be shared, there is no need for dividing parts of the application according to functionality or purpose. When an application starts growing in complexity, however, there comes a time where splitting the application into multiple parts would make development and maintenance easier.

While the Meteor guide does propose a few approaches for splitting a system into multiple applications, they don't really compare to what is usually considered a microservices architecture. For instance, the Meteor guide suggests deploying the same application onto multiple servers, differentiating between them with the use of settings. When it comes to sharing common code between two different applications, it is suggested either to copy the code as is to both or to publish the functionality as a package to be imported by both applications. [23]

The one thing that the guide does explain in detail is how to share a common database between multiple applications, be they the same application deployed onto different servers, or completely different applications. [23] As this is possible, the concept of having a proper microservices architecture using multiple Meteor applications should be possible in theory. While there exist a few open-source proof of concept projects around the subject, it seems microservices architecture applications with Meteor have yet to gain popularity.

3.3.7 Hosting Options and Tools

As Meteor is based on Node.js, any application developed with it can be hosted anywhere, just as Node.js applications. However, there are a few main options that are popular in the Meteor community.

Galaxy. Galaxy is a service built by MDG, created specifically to run Meteor applications. Is it a distributed system based on Amazon Web Services (AWS), and as such offers services such as automatic scaling, production debugging tools, application analytics, and Meteor-specific help. [24] Galaxy pricing is based on the size of the application and its container, starting from 7$ per month for a 256MB instance of 0.3 EC2 Compute Units (ECU) [25]. As Galaxy provides the hosting server, the user only has to set up their database hosting server or service.

Meteor Up. Meteor Up, commonly known as "mup", is a third-party open-source tool for deploying Meteor applications to any server over SSH. It automates the manual steps otherwise involved in using Meteor's build tools and handles moving the application bundle to the hosting server. [24] It requires the user to handle obtaining the hosting server, as well as the database hosting server or service. Additionally, the user has to take care of scaling, load balancing, application analytics, and other such services on their own. In essence, Meteor Up creates a number of Docker containers necessary to run a Meteor application.

Docker. Meteor applications can also make use of container-based deployments. While Meteor Up provides some Docker images automatically for the user, these can also be picked manually, or custom-created for more control over the instances. There exist several ready-made Docker images that can be freely used. [24]

Custom Deployment. Meteor applications can also be deployed and hosted completely from scratch, making use of Meteor's build tools. The build tools provide the user with a plain Node.js application bundle, which can then be set up wherever the user might want. [24] This type of deployment requires the most manual work and configuration, which in turn gives the user more freedom to choose their hosting service provider and customise their production pipeline.

4 Technical Debt in Software Development

This chapter explains the concept of technical debt in software development, what it means in terms of application development, as well as what special considerations come from dealing with modern technologies. It also explores different strategies for getting rid of and preventing technical debt in a software project.

4.1 Software Development Life Cycle

The Software Development Life Cycle (SDLC) is a term used to describe the process of building and maintaining software systems. It often comprises various phases, from preliminary design and development to testing and evaluation. There are two main SDLC methodologies used by developers: traditional and agile development. [26]

Traditional SDLC. Traditional software life cycle methodologies, such as the Waterfall methodology, are based on a sequential series of steps. As we can see from Figure 8, these steps often include mapping the requirements, planning and designing the architecture, developing the software, testing, and deploying it to production. The main idea of such a methodology is to approach each step one at a time, and finishing it before moving on to the next. This means all project requirements should be well-known from the beginning, and shouldn't change during the lifespan of the project. Done with such an approach, the software is usually delivered only once completely ready, often leaving a long gap between the beginning and the delivery of the project.

Agile SDLC. Agile software life cycle methodologies, such as SCRUM, are based on the idea of incremental and iterative development. In this approach, instead of going linearly, each step in the life cycle is revisited during each iteration, as showed in Figure 8. This enables the delivery of frequent features, or increments, to the software, as well as rapid feedback from customers. This approach is well-suited to projects where the desired end result is unclear, requirements are likely to change, or where the software can already be used even in a partially finished state.

Technical debt is a common occurrence in any software development project, and it can be introduced in different stages of the SDLC, regardless of the selected methodology [27]. In traditional SDLC, a common cause of technical debt might come from outdated project requirements. In agile SDLC, the pressure to deploy a new software increment might cause the development team to make poor code design decisions, trading longer-lasting high-quality code for a faster implementation time.

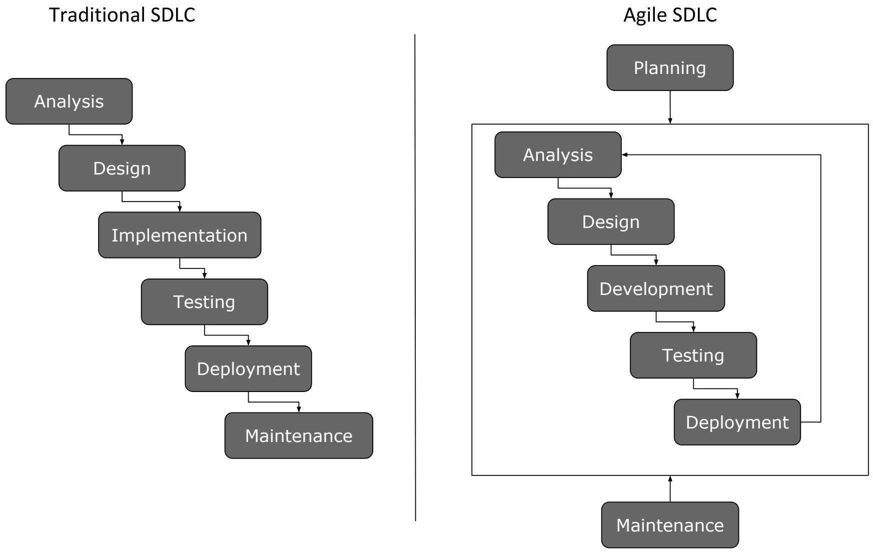


Figure 8: Traditional and agile SDLC.

4.2 Technical Debt

Technical debt is when long-term code quality is traded for short-term gain. Often it is attributed to shortcuts and workarounds in the source code of the software, where developers choose quick and messy implementations instead of spending time on code quality and maintainability. Whereas this approach is good for quickly pushing out new features, it generates more future work on bug fixes and rewrites.

The term technical debt used to be something strictly technical, closely related to code quality, and often something only the developers see and care about. Since its introduction, the term has evolved to encompass much more than its original meaning. Nowadays, it is often used to describe also a lack of documentation or specifications, a lack of tests, poor architectural decisions, or a messily structured codebase, as seen in Figure 9. [27]

4.2.1 Code Quality Debt

As one of the most often referenced types of debt, technical debt in code quality is one of the most commonly occurring types of debt. As mentioned earlier, developers might make the conscious or unconscious choice of sacrificing code quality for faster implementation of features or to get a faster time to market with their software.

Code quality is not only about making poor code design decisions. The lack of systematic testing for catching breaks in functionality, be it manual tests or automatic test suites, is one of the main reasons code quality might suffer. In addition to contributing to poorer code quality, the lack of existing

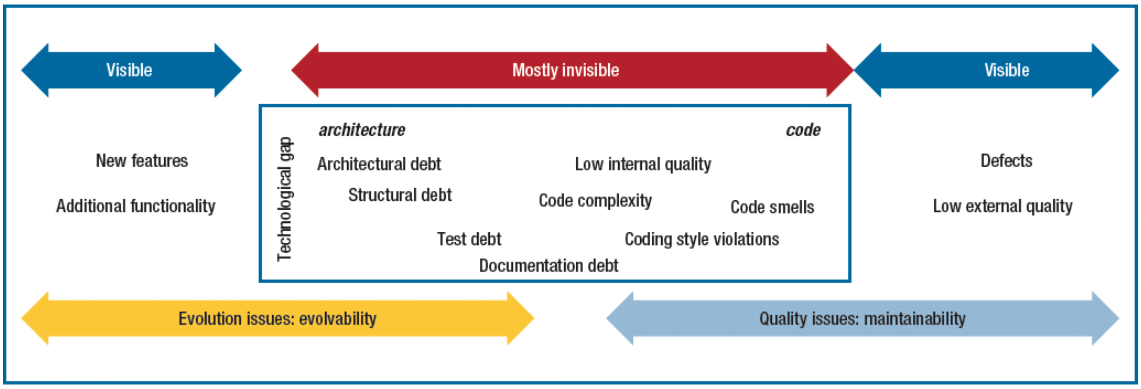


Figure 9: Technical debt landscape. [28]

tests, or with a high enough coverage, can be seen as a lack in software quality in itself as well. [28]

4.2.2 Architectural Debt

Architectural debt is often encountered in software projects that have had a longer life span, including many changes or additions in requirements or user base. When the initial architectural design, or lack thereof, doesn't keep up with the changing software and its requirements, some new architectural decisions and plans should be made [28].

For example, a development team might start developing a small monolithic application for a select group of users, and later on, new requirements come along that require the application to be scalable in certain parts and not others. To make the application meet the new requirements, some restructuring of the application and its architecture is required if the development team wants to avoid complex development decisions and implementations.

4.2.3 Non-Functional Debt

Non-functional qualities are often incorporated in the concept of code or architectural quality, but as far as analysing debt goes, it might be beneficial to look for debt from a non-functional point of view. As the term covers so many different attributes over different categories of debt, it might as well be considered its own category.

Often non-functional requirements should be taken into account already in the beginning stages of a software project, as they are harder to add in after development has already been underway. For example, if things such as application security or integrity are not designed, they might be close to impossible to implement at a later stage, at least without a considerable amount of additional work and refactoring.

The lack of some non-functional attributes can more easily be remedied in a software project than others, and can thus be dealt with as well as any other type of debt. Often the answer to bridging non-functional debt can be found within the answers for dealing with other categories of technical debt. When it comes to dealing with issues in scalability or availability, the solution often has to do with architectural actions. On the other hand, in terms of performance or maintainability, the necessary measures can be closely affiliated with code quality actions.

4.3 Dealing with Technical Debt

As stated earlier in this section, technical debt can cover a lot of different areas, from lack of documentation to architectural design flaws. For the purposes of this thesis, dealing with technical debt will focus on two main types of technical debt: traditional, or code quality, debt, and architectural debt.

4.3.1 Code Quality

While technical debt is practically unavoidable, there are different approaches and practices that can be employed to prevent or substantially diminish it. These approaches can be roughly divided into two categories:

1. Clean up
2. Prevention

While the first category focuses on dealing with the damage after it has already been done (i.e., cleaning up the codebase) the second one deals more with continuous efforts to prevent it from coming to life in the first place.

Dealing with existing technical debt in any software project requires an active approach and conscious decision to do so. If the technical debt in question has more to do with the code quality than anything else, code refactoring may be used, either incrementally or during a longer period of time. For example, some agile teams may employ refactoring sprints, focusing solely on improving the code quality instead of writing new features. More often than not, it is difficult to justify using a longer period of time on something that brings no immediate visible value to shareholders. In these cases, it is easier to integrate some refactoring tasks with the implementation of new features, thus improving the existing codebase at the same time as new features are developed.

There are a few different approaches that can be used when cleaning up a messy codebase.

Parallel Rewrite. In this approach, the whole application is redeveloped in parallel to the original one, making sure that the application structure and code are up to the required standards. This is a time-consuming and laborious process, and is rarely worth the effort and time it would take to rewrite the whole application.

Partial Rewrite. Instead of rewriting the whole application, only selected parts are chosen to be rewritten. These might just be updates and refactoring done to the code itself, or an effort to completely separate these parts from the original application, e.g., when separating business logic parts of a monolith into microservices.

Phaseout. If technical debt is not dealt with actively during the software development life cycle, it may eventually lead to unmaintainable code, and to having to abandon developing the project altogether in favor of starting or switching to a new one. In these cases, some changes might be required to guide the users to the new service while waiting to finally discontinue the existing one.

A more sustainable approach to dealing with technical debt is to try to actively prevent it from accumulating. There are many different ways to achieve this, mostly having to do with the habits and practices of the development team itself.

Code Reviews. By increasing the amount of people involved in evaluating any new code submitted to the project, the quality of the code itself should be higher than if only the developer in charge of writing it were involved. Not only is there a pressure on the developer to submit only high-quality code, the other developers may also have higher standards of the quality required, or just be able to catch some segments of low-quality code that might have gone unnoticed otherwise.

Acceptance Criteria. By having some collectively agreed-upon guidelines and criteria of what is required of any code submitted to a project, the overall quality of the project should stay at the same level of standards. The criteria should be set at a level the team can commit to; if the bar is set too high, it is likely that one or more members or the team will abandon the criteria, rendering it useless. If it is set too low, however, it will do nothing to help the quality of the project.

Workflow. Having a standardised workflow can greatly improve the quality of a software project. While a workflow in itself isn't a guarantee of high quality code, the elements integrated in the workflow are what can make a difference. These can be any good standard practices, such as the code reviews and acceptance criteria mentioned above, or something more technical, such as git branching techniques, or test writing practices. The main idea is again to standardise the code coming from different developers in the team, in order to standardise the codebase itself.

Automatic Tests. There might not always be the resources to manually go through every code change, and even if there were, it would not be ideal or sustainable in the long run. This is where automatic tests come in, with the intent of moving the responsibility of checking the code quality from the developers to machines. Of course, written tests can only do so much, and are usually used to check for functionality breaks, or linting errors. Anything more complicated or opinionated should still be checked by developers, if there are the resources to do so. Still, any automated tests are better than none at all, and they work best when integrated into the work or deployment flow, such as in continuous integration pipelines.

While any one of these approaches may help in keeping the codebase at a high quality and preventing the accumulation of technical debt, usually a number of these techniques are used in unison. A high quality workflow usually comprises the automation of code quality checks, making it as easy as possible for developers to focus on generating content instead of doing manual code reviews.

4.3.2 Architecture Quality and Migration

While code quality is something that can be worked on incrementally, a bigger task to take on is refactoring or redesigning architecture choices, or re-engineering a whole application. As the software architecture covers the whole application, any changes may have an impact on any part of the application. Some reasons for refactoring the architecture are, e.g., simplification, modularisation, or improvement of the program structure. Another driving factor might be the preparation of the program for transformational activities, such as moving the system into a new environment without introducing changes to the functionality of the program. [29]

There are a few different approaches when it comes to dealing with architectural debt.

Program Restructuring. The removal of anomalous, redundant or dead code, as well as the separation of business logic from data, are considered in program restructuring. In addition, the source code could be checked against new quality standards, and changes made accordingly. [29] While this is closely related to code refactoring, program restructuring focuses on the overall code structure of the application, thus directly affecting the architectural quality of the software.

Architecture Transformation. In a complete architectural redesign, the current architecture is extracted, and a new target architecture designed. The transition from the current architecture should then be planned and implemented. Transformation is usually a much bigger task than restructuring but might lead to more long-term benefits. [29]

Lift and Shift Migration. The shifting of application language, platform, or data migration without any functional changes is considered a lift and shift migration. For example, migrating a monolithic application to a distributed platform, or moving an on-premise application to a cloud provider can be regarded as lift and shift migrations. Such a type of migration is rarely a standalone process and is often combined with the aforementioned program restructuring or architectural transformation, either before, during, or after the migration process. [29]

All of these architectural refactoring processes are usually quite extensive, and take a lot of resources and time to plan and implement correctly. These types of processes are usually applied only after careful consideration and evaluation of the value entailed.

4.4 Special Considerations in Modern Technologies

As modern technologies tend to have faster development rates than already established technologies, technical debt tends to accumulate more rapidly than in other projects. For example, a technology in its first versions might introduce breaking changes more often than older technologies, who already have an extensive user base that they need to take into account. This means that to keep up with the new versions, the application must go through many refactoring stages, or risk being left to continue development with a technology that is no longer supported or even safe to use.

In some cases, a promising modern technology might meet with a surprising end of life, in case the technology is acquired by another party and discontinued, or given up in favor of a complete rewrite or other strategic decisions. In these cases, the developer is left with the decision to either keep the current application in the condition it is in, with no more than maintenance tasks possible, or to shift the whole application onto a new technology, if possible.

On the other hand, choosing a modern technology instead of one that has already been around for a while might have its own benefits in regard to technical debt. Often the technology itself, or the tools associated with it, might have accumulated their own share of technical debt. In these cases, the technologies and tools used might suffer from delays in delivering newer features due to time spent on dealing with technical debt accumulated over the years.

5 The Cloud Environment

This chapter gives an overview of the cloud environment, how it has developed over time, as well as of the technologies required for the modern cloud to be born. It also presents the most popular public cloud providers and analyses what they have to offer in terms of services for web applications.

5.1 Web Hosting Evolution

Before the generalisation of cloud computing, web application hosting and servers were much less sophisticated than they are today. As more work and costs were spent on operations and maintenance tasks, any updates, fixes, or new applications required a lot of joint effort from different teams to be made possible.

5.1.1 Hosting Services

Along with web technologies, different hosting options have come a long way since the beginning of the modern web era. Web hosting as a concept became popular around the time of the launch of Geocities in 1994. Before that, hosting a web application usually meant having your own computer or server for hosting the application. Geocities was among the first to offer commercial server space for hosting applications. [30]

Little by little, shared hosting service providers started generalising. These early-day providers were selling a very limited amount of storage space, along with data transfer packages, from their servers.

Along with shared hosting services, the need arose for dedicated hosting services, especially for bigger businesses with busier websites. Dedicated hosting services offered the option for their users to have partial to full control over the offered servers, depending on the needs of the business in question and the service provided. This gave business users the control they needed to grow their services and their user base along with them.

5.1.2 Private Data Centers

Often, the control offered by these service providers was deemed insufficient, especially by bigger and stricter organisations and businesses. Even with multiple rising service providers, as well as an expanding choice of hosting services, these enterprises would still choose to build and maintain their own data centers instead of paying for a hosting service.

Most early hosting services simply didn't have the capacity to offer enough servers for larger businesses, and eventually, they didn't need to. Large enterprises would invest and build their own computer centers, dedicated to hosting and running all their services. By choosing to have their own data centers, however, businesses accumulated growing costs surrounding acquiring, upgrading, and maintaining infrastructure.

5.2 Virtualisation and Containers

Virtualisation is the concept of making a virtual image or version of a server, operating system, storage device, or network resource, so that they can be used on multiple machines at the same time, or use many instances of different machines on one physical machine. [31]

The development of virtualisation and containerisation technologies played a crucial part in contributing to what we know as cloud computing today. As different levels of virtualisation exist, so can multiple services based on what they can offer.

Virtual Machines. Virtual Machines (VMs) are an application of server, or hardware system, emulation, based on hypervisor virtualisation [32]. Each VM comes with its own operating system, with respective binaries, libraries, and applications as needed. Virtual machines are what make it possible for a single physical server to act as if it was split into many servers while sharing its physical resources between them.

Containers. As virtual machines cover the hardware aspects of computing, containers are the application of virtualising the operating system level. This type of virtualisation is usually called containerisation, as it uses so-called containers to emulate operating systems. [32] As containers don't have to worry about emulating the hardware level, they are much smaller in terms of size, and generally a more lightweight approach to virtualisation, as we can see in Figure 10.

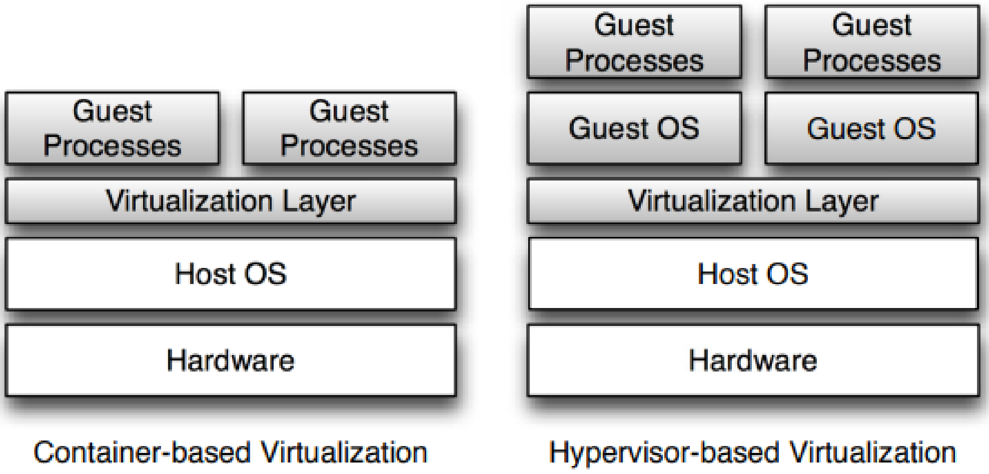


Figure 10: Container-based and hypervisor-based virtualisation. [32]

Virtualisation can be thought of as a prerequisite of the modern cloud as we know it. While some hosting providers can deliver dedicated machines as a service, cloud providers usually rent out a part of their cluster of machines. In this server model, it means that, e.g., many different operating systems can co-exist on the same physical instance at any given time. Therefore, virtualisation is a must if all resources are to be utilised to their full potential.

Operating system virtualisation is not the only prerequisite of the modern cloud. Since cloud providers are renting out memory, processing power, or anything the customer requires, at a highly scalable level, hardware resources must also be shared efficiently between the machines of the data center. As it is, the cloud is essentially a pool of computers sharing their resources thanks to virtualisation technology.

5.3 The Cloud

The concept of cloud computing, as well as cloud hosting and other various services, introduced a change in how infrastructure was conceived. Instead of owning physical servers and infrastructure, organisations could now lease shared or dedicated resources from server clusters, or cloud providers, on-demand.

5.3.1 Private Cloud

Although the advantages of the cloud have been emphasized since their generalisation, the shift of enterprise-level services from self-hosted servers to the cloud has been relatively slow. This is mostly due to security concerns, as the physical layer of isolation isn't handled by the server users themselves anymore, but instead left to the responsibility of the cloud provider.

These security concerns, among other things, gave way to the concept of private cloud. This meant that the same concepts of cloud computing were implemented, only on privately owned hardware. These private clouds are essentially the continuation of early data centers, but with the added benefit of cloud computing technology, such as hardware virtualisation. Despite these efforts on behalf of private owners, most of the existing private clouds can't measure up with the competing public cloud providers in terms of scalability, stability, coverage, and sheer size.

5.3.2 Hybrid Cloud

As cloud services continued evolving, even private cloud users started to shift towards the public cloud, at least in part. Due to migration difficulties, existing concerns, or just the reluctance of moving their services, a lot of private cloud users ended up with a mix of private and public clouds.

A big factor in this shift, besides receding security concerns, was the affordable pricing of the public cloud. On top of enterprises not having to pay for their own dedicated infrastructure, cloud service providers could offer bulk prices, allowing enterprises to scale at a previously unprecedented rate with much lower costs. [33]

As services were now in both private and public clouds, there was a need to bridge the gap between them to allow services to communicate with each other. This gave birth to a merged implementation of both private and public clouds, also called hybrid cloud. Essentially a tailored service from public cloud providers, a hybrid cloud allows organisations to keep their services in both private and public clouds, but still work and communicate as if they were in the same environment.

The term hybrid cloud is also used to describe a multi-cloud environment, even if no private cloud is involved. In these cases, the services live on multiple public cloud provider environments, instead of sharing both private and public clouds.

5.3.3 Public Cloud

Perhaps the most commonly thought of cloud, the public cloud is what is usually meant when the term cloud is mentioned. Public clouds are managed cloud services and components that are for sale, either in large quantities or as single services.

There are currently three main large public Cloud Service Providers (CSPs) with smaller ones being born every now and then.

Amazon Web Services. Amazon Web Services (AWS) is a subsidiary of Amazon that provides a large range of cloud services. Launched in 2006, it is the oldest of the CSPs mentioned here, and with the widest selection of services. Due to this, it is currently the most popular CSP in use. [34]

Google Cloud Platform. Google Cloud Platform (GCP) is Google's set of cloud services that run on the same platform as Google's own services, such as YouTube or Google Search. GCP services have been generally available since 2011, but have gained more popularity only in the last few years. [34]

Microsoft Azure. Microsoft's Azure Cloud, commonly known as Azure, is Microsoft's cloud service provider. It has been around since 2010, when it started out as Windows Azure, later changing its name to Microsoft Azure in 2014. [34]

All of these, as well as smaller CSPs, offer a wide range of services, some more tailored than others, but many overlapping in their essence. To differentiate between different CSPs, a closer look should be taken at their offered services, implementation, pricing, geographical locations, and security implementations, to name a few.

5.4 Cloud Services

In contrast to traditional hosting service providers, CSPs offer a wider range of options to choose from when it comes to hosting web applications. While the traditional server space leasing still exists as infrastructure as a service, more tailored solutions have risen up beside it as well. Instead of simply offering virtualised infrastructure space, cloud providers can offer a platform for hosting web applications, space for running containers, or even just functions.

As mentioned earlier, virtual machines and containers are the building blocks of the main computing services offered by public CSPs. With this existing shared resources model, users can freely choose to customise what they buy according to their needs, be it memory size, efficiency, performance, or anything else the cloud provider might have offered to be customised.

There are a few common types of cloud services that are offered by most CSPs that differ in terms of how much responsibility and access is given to the users, and how much is kept in the CSPs realm of liability. The differences between management responsibilities can be seen in Figure 11.

Infrastructure as a Service. One of the earliest forms of cloud services, Infrastructure as a Service (IaaS) offers the equivalent of running virtual machines or servers on the cloud. Users may choose from the technical resources needed, such as memory size and storage space, and are responsible for the infrastructure, as well as anything they might want running on it. [35]

Platform as a Service. One step further in the chain of managed services, Platform as a Service (PaaS) offers a managed infrastructure for their users, letting them focus on the application itself. Here the users don't have any access to the infrastructure but are responsible for the application hosted on it [35]. Such a service is often used for web applications, where the infrastructure needed is abstracted behind a platform service, making it easier for the user to focus on the actual implementation instead of operational tasks.

Function as a Service. A rising trend in the cloud computing world is to divide application logic into smaller and smaller pieces, similar to the idea behind the microservices architecture. This has led to another popular service of CSPs, which is using machine resources for just the length of running a single piece of code or function, known as Function as a Service (FaaS). FaaS can be seen as a subcategory of PaaS, but as it has gained popularity especially in serverless applications, it is mentioned here separately. Popular examples of FaaS include AWS's Lambdas, GCP's Cloud Functions, and Microsoft Azure's Functions.

Software as a Service. Software as a Service (SaaS) is a term used to describe applications that run on cloud infrastructure and are accessed through a client, such as a browser. Users of SaaS don't usually have any kind of access to the infrastructure itself, or any means to manage it. Such applications include, e.g., email service providers, or social media platforms. [35]

Database as a Service. Managed databases, sometimes referred to as Database as a Service (DBaaS), is software that enables users to set up a database on cloud infrastructure without having to worry about its technical implementation or requirements. [36] The technical implementation is often abstracted behind a simple interface. DBaaS can be considered a subcategory of SaaS.

Kubernetes as a Service. Many CSPs offer containers as a service as well as virtual machines, and often containers are managed using a container orchestration system, such as Kubernetes. This has led CSPs to go one step further to facilitate container orchestration and provide Kubernetes as a managed service. This allows users to easily manage, scale, and customise a cluster of containers on a CSP. Examples include GCP's Google Kubernetes Engine (GKE), AWS's Elastic Container Service for Kubernetes (EKS), and Azure's Kubernetes Service (AKS).

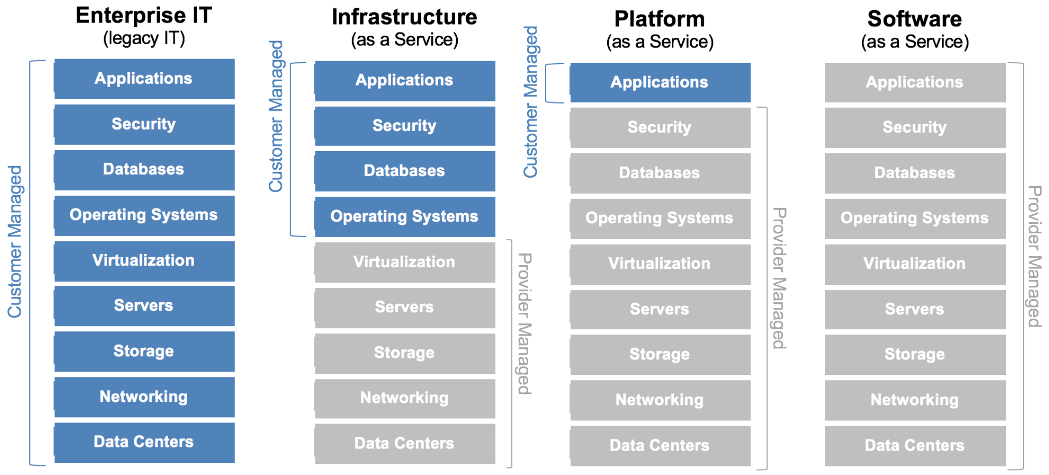


Figure 11: Management responsibility differences between traditional IT, IaaS, PaaS and SaaS. [35]

There exist many more tailored and specialised services provided by CSPs, but for the purposes of this thesis, we will limit them to the ones presented in this section.

5.5 Benefits and Drawbacks of CSPs

There are a lot of reasons why services and organisations are moving towards the public cloud, while there still exist some causes for rethinking a move to CSPs. Generally, the benefits of cloud computing seem to outweigh its drawbacks. Nonetheless, every use case should be analysed separately to see if a cloud environment is the best approach.

Out of the claimed benefits of a cloud environment, here are some of the most commonly mentioned ones.

Price. Since large CSPs have a well-established base, it makes it easy for them to have low prices on their services, without having to add any additional costs to users. Users will only pay for what resources they use, and the payment model is often pay-as-you-go, meaning no long-term commitment is needed. [31]

Services on Demand. Through CSPs, users can quickly and effortlessly set up new virtual machines or services in practically no time at all, without having to contact a vendor or go through a lengthy process to get them set up. The same goes for shutting down services, as users have full control over the management of their allocated services. [31]

Location. As large CSPs have data centers set up all around the world, users may set up their services wherever they wish, or to multiple locations at once [31]. This is a great advantage when compared to a private cloud, for instance, as the cost of maintaining just one data center is already high, but the cost of having multiple ones set up would be even higher.

Security. Although security is a debatable matter, CSPs have a long history of scaling up their security requirements when it comes to cloud infrastructure. While application security is still left in the hands of the user, things such as network security and physical security are handled by the CSP, who is accountable to certain standards, which in many cases are much higher than the ones users might have.

While a cloud environment does have many advantages, there are a few risks to take into account when moving towards a CSP.

Access. While restricting management access is one of the services provided by CSPs, there are some cases where a user might want to have full management rights to the resources their services are running on. Often the offered management services are sufficient, but full access, e.g., physically, is not something that CSPs can offer.

Security. While CSP's security standards are generally high, some organisations' standards might be even stricter, leading them to have higher standards than what CSPs generally have to offer.

Location. Some regulations and laws, such as the General Data Protection Regulation (GDPR), require certain information to be kept within a certain geological location. Some cloud services might not be able to provide all of their services in such locations, or might not even have data centers available there, meaning that sensitive information might go outside of these bounds. In these cases, these cloud services are not usable.

Vendor lock. Not a risk of cloud environments in itself, vendor lock is a situation where it becomes nearly impossible or extremely difficult to move away from a current CSP. Nowadays, a lot of cloud services are ambiguous in terms of CSPs and can be shifted more easily than before between them. Still, some services only exist on certain CSPs and must either continue to do so or find different solutions if it is no longer the desired situation.

Learning curve. As a constantly evolving environment, new services are developed and published by CSPs at a rapid rate. For an inexperienced developer, getting to know the public cloud ecosystem might present a time-consuming challenge. As services might vary from one vendor to another, mastering one ecosystem might not guarantee that the rest will also be mastered as quickly.

6 Case: Project WorkTracker

This chapter presents an hour-tracking application, its main functionalities, as well as its technical stack and implementation. It also takes a look at its known technical problems, as well as at the goals hoped to achieve during the migration process.

6.1 Application Description

The goal of the application was to create an internal work time tracking software for the employees of the company to facilitate payroll events. The minimum requirements were that internal employees would be able to mark and submit their work hours to payroll. The main goal was to also have all external employees using the same time tracking system.

6.1.1 Features

The application was designed and implemented based on the following features.

User information. Basic user information, such as name, email, supervisor email, organisation, team, and password, had to be saved into the application. These would only be visible to the users themselves and the administrators.

Invitation-only login. The user base of the application had to be limited since it was a company-internal software, and outsiders shouldn't be allowed into the application for security reasons. Registration by an invitation-only protocol was decided upon, which would allow the administrators to send out invitations for the target users.

Work hour tracking. Probably the main functionality of the application, the ability to add, edit, and remove daily work hours was required. Apart from tracking normal work hours, special payroll significant information, such as sick leave and vacation days, should also be possible to add.

Tags. The ability to create and use tags for different work hours was a requirement. This was mostly meant for payroll administrators to be able to analyse and create meaningful reports based on the employees' work hours.

Roles. Different user roles, representing different levels of rights and access, had to be included in the application. These included roles for employees, supervisors, and administrators. Also, limiting information based on the role of a user was a requirement.

Filtering tools. All users should be able to have basic filtering tools for their own work hours. On top of this, supervisors and administrators should have the possibility to filter more than one employee's work hours, as well as to generate reports based on the filtered results.

Time reports. Automatically generated time reports to be sent out monthly were to be possible within the application. These time reports would be sent out to and accessed by the corresponding supervisors only. There should also be a way to approve or reject them.

Approval and locking hours. Once a certain time report had been approved, the employee shouldn't be able to edit or modify their working hours for that period of time, unless the report had been rejected by the supervisor.

Teams and organisations. As the software was to be used for external employees as well, additional information about the organisations, or companies, and teams was required, as well as a way to manage this information.

Company limitations. As different work hours and tags for different companies were to be in use, it was required that the tags used to mark work hours should be company-specific, as they could potentially include sensitive information.

Assignment tags. To differentiate the use of regular work hour tags and billable projects, the concept of assignment tags was introduced. This would be a separate tag visible to internal employees only and would contain information about billable projects.

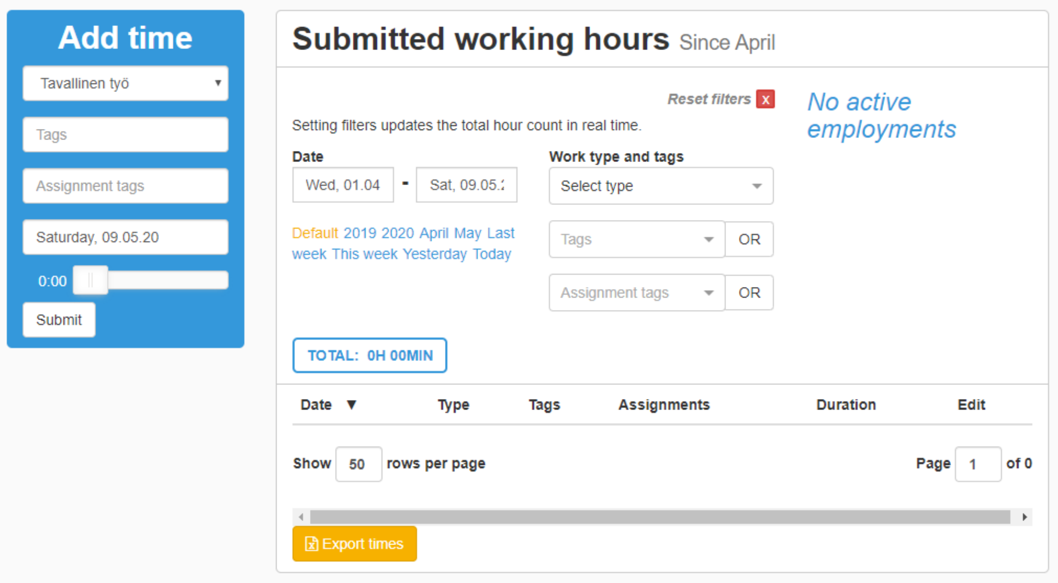
Employment details. The software would be used to ask for and store critical employment details of internal and external employees. These would include details needed for employment contracts, as well as information about past and current projects if the employee had been in multiple assignments.

6.1.2 Views

The application consisted of a few main pages, or views, that each held their own functionality. Here is an overview of the different pages of the application, as well as of the functionalities they offered.

Time entry. The main page of the application, this was the page where all users would submit their work hours, as shown in Figure 12. It consisted of the time entry form, some basic filtering tools, as well as a table of previous time entries made by the user. The table provided the functionality for editing or deleting a single entry, each on its own row.

Basic info. This was a user's personal information page. Here was a form where they could see information about their team, organisation, and supervisor. It also provided the functionality for updating a user's name or email address.



Submitted working hours Since April

Saturday, 09.05.20

0:00

Setting filters updates the total hour count in real time.

Wed, 01.04 - Sat, 09.05.2

Select type

No active employments

TOTAL: 0H 00MIN

Date ▼	Type	Tags	Assignments	Duration	Edit

Show 50 rows per page

Page 1 of 0

Figure 12: Time entry page.

Team info. If a registered user was also the supervisor of a team, a page containing their team's information was available to them. It contained their basic information, as well as their time entries for the ongoing month.

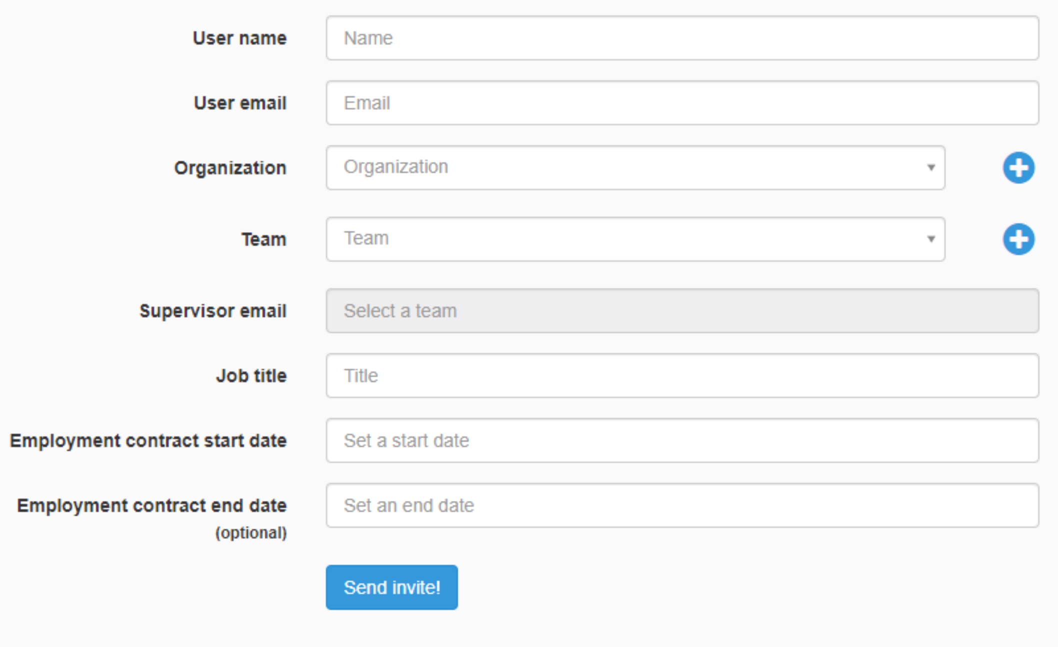
Invite and register. The main tool for adding users to the application, this page consisted of an invitation form requiring some basic information about the upcoming user, such as their email address, as seen in Figure 13. The same page provided the functionality for invited users to register into the system, provided their invitation link was valid.

Tag management. This page provided a tool for managing the tags that users could create using the time entry form. As there would eventually be tags that needed to be removed, or some duplicate tags that should be merged, this was where the tool for that functionality resided.

Teams and organisations. This page provided an overview of all the different teams and organisations registered in the application. It also provided the tools for adding, editing, and deleting them, as well as managing their corresponding supervisors.

All users. This page showed all of the users registered in the application, as well as their basic information, team, and organisation. It also had the tool for showing their employment information, as well as their saved contract information. Different roles and permissions were administered through this page, as well as deleting a user from the system.

All times. This page contained all of the time entry info for all of the users registered in the application. It also contained extensive filtering tools, as well as the possibility to extract the time information into a csv-file.



User name	Name
User email	Email
Organization	Organization
Team	Team
Supervisor email	Select a team
Job title	Title
Employment contract start date	Set a start date
Employment contract end date (optional)	Set an end date

Figure 13: User invitation page.

All time reports. This page had a collection of all sent time reports for each user in the application. This page allowed the administrators to have an overview of which time reports were accepted, and which were yet to be acceptable, i.e., in a problematic state.

Single time report. A page containing a single time report was sent at the end of the month to each user's corresponding supervisor. If a team contained more than one person, the report held time information for the whole team.

6.1.3 User Roles and Permissions

The different user roles implemented in the application were crucial in limiting the information conveyed and the tools available to each user. Here is a short summary of the different roles, as well as their respective rights and views.

User. The most common user role, a plain user had the right to submit and edit their own work hours, create work hour tags and change their basic information.

Team admin. A team administrator had the same rights as a basic user, on top of which they could see their team members' information, as well as their work times.

Admin. An administrator had the right to every user's information and work hours, as well as the right to invite and remove users, add teams and organisations, and manage employment information as well as time tags.

Supervisor. A supervisor only had the right to see their respective team's work hour report at the end of each month, as well as approve or reject it.

6.2 Technical Description

At the beginning of the development of the application, the technology choices, as well as the architecture of the application, were decided with the development team. Apart from a few minor libraries, no major changes had been made to these during the life cycle of the application so far.

6.2.1 Technology Stack

The modern web framework Meteor was chosen as the technical platform for the project implementation, due to its making use of only one programming language (i.e., JavaScript) throughout the stack. At the beginning of the project, in 2014, Meteor was at version 0.9, one version away from its first major release. At the end of its active development, the project had updated the version of Meteor to 1.4.

As for the project database, the non-relational database MongoDB was chosen, due to its close pairing with the Meteor framework. It was the only viable option at the time, since using any other database would have required a lot of manual work, and wasn't deemed worth it.

The codebase was hosted on GitHub's Enterprise plan repository, which allowed for the codebase to be private within the enterprise that owned it. Having the codebase available publicly would have been a security risk, especially during early development days.

6.2.2 Development Practices

The development team was quite small, consisting of three people. Due to the small size of the team, not many development practices were enforced, as they would have slowed the development of new features down considerably. The methodology used in development was an agile one.

In this case, the team chose a faster delivery of features over high-quality code. This meant the project had practically no implemented tests or any automated pipelines for integration or delivery. There were, however, some code reviews, especially when introducing a new member to the project team. This lowered the number of bugs accidentally introduced into the codebase somewhat.

6.2.3 Architecture and Design

The application was written following a monolithic architecture approach. The codebase was unified and hosted on a single repository, and a lot of the application code was reused throughout the project. Especially in the frontend, a lot of templates were used to provide a basic view for any application users. These templates were then reused with administrator additions, to provide more information and tools where needed.

The application itself relied heavily on reactive data, expecting near real-time updates to application data throughout the clients. While this was not a requirement in the daily use of an average user, the tools that the framework provided made it simple to build the application on top of this principle.

The application followed the design approach of the Meteor framework, following mostly an SPA design. While the application seemingly fosters many pages and views, it actually uses a routing library to simulate different pages, making it seem like an MPA.

6.2.4 Hosting and Deployment

The project was deployed with the package Meteor Up (mup) and hosted on a dedicated server instance in UpCloud [37]. Meteor Up was an automated build tool that used Docker containers for building and running Meteor applications. In this case, Meteor Up created four Docker containers upon deployment:

1. **application-container**. The main container for the application, it held the JavaScript code bundle containing all application logic.

2. **application-container-reverse-proxy**. The reverse proxy front for the application, all traffic to the application was routed from here.

3. **ssl-certificate-container**. Since we wanted the application to use https, a separate container was brought up to automatically renew the application's SSL certificate every three months.

4. **mongodb-container**. The container for the database, this was the automatically generated local container for the MongoDB database.

6.3 Technical Problems

At the beginning of this project, the application hadn't been updated or developed in a few years, outside of a few minor bug fixes. However, there was an extensive list of more general technical problems that were waiting to be tackled and would be beneficial to be taken into consideration prior to or in parallel with any modernisation or migration attempt.

Outdated codebase. As the project was started in the early phases of Meteor, a lot of the code quickly became outdated, either in conflict with the proposed approaches of newer Meteor versions or just plainly harboring known security risks. Even with gradual updates to some parts of the application, this meant that the codebase included code that was supported in versions as early as Meteor 1.3, whereas at the time of this project Meteor had just released its version 1.9, with many breaking changes in between.

Updating the application. During each new code update, the application had to be manually updated using Meteor Up's tools. The updates each lasted up to 5 minutes, and the application was unavailable during that time. The deployment process also required that the deployment machine had access to the hosting server, usually through SSH keys.

Uneven server load. Most of the time, the application was at quite a low level of usage, as users would report their work hours maybe once a day or once a week. However, once a month, when the time reports were to be sent, the server load grew considerably, as there were many calculations and processes involved in sending these reports. Since the application was hosted on only one server and could only be scaled vertically, this resulted in the application server being widely oversized in regard to the application usage, except for those once a month occurrences.

Adding new features. As the application architecture was monolithic, a lot of code was entwined, and dependencies were hard to follow. This meant that developers needed to have a good understanding of the application as a whole before being able to add or remove a separate feature.

Slow application. Due to Meteor's built-in reactivity, users with more complicated work hours needed to have more data on the client, which led to the application slowing down considerably. Similar problems were noticeable on the backend, where the logic for one simple task was found within a function responsible for many more things than just the task itself.

Database instability. The fact that the production database was hosted in a local Docker container with no replicas or backups was a huge risk, as it meant that if the application server were to crash into a state where it couldn't be repaired, the whole application data would potentially be lost.

6.4 Goals

The main goal of this modernisation project was to tackle sufficient technical debt as to make future development easier. This included dealing with the technical problems mentioned in the previous section. The secondary goal was to ensure the integrity, scalability, and availability of the application, as well as to make life easier for its future maintainers. These goals can be summarised below.

Maintenance and updates. A new update strategy was strongly desirable, as the current one included a lot of manual work, as well as application downtime. The goal was to simplify the updating process, as well as reduce the application downtime to near-zero, improving the application availability in the process.

Infrastructure optimisation. Since the current infrastructure, as well as the application architecture, limited the scaling options to vertical scaling, the cost of having a hosting server used at minimum capacity most of the time was something to be improved. The goal was to have an infrastructure solution that would be optimal in normal use times, as well as be able to serve during high load times, such as the end of the month.

Flatten the learning curve. Due to the state of the codebase, introducing new members to the development team was a difficult task, as a thorough knowledge of the development history of the application was required in order to make sense of the application. At the same time, introducing new features to the project was also hindered by the same reasons. The goal was to make it easier for future developers to learn and contribute to the project, and make the codebase similar to the suggested application structure of the newer versions of Meteor.

Technical optimisation. Since not all code introduced to the project had been peer-reviewed, there were a lot of pieces of code that were performing suboptimally. The goal was to make the application run faster by optimising low-quality code, including anything from unnecessary data subscriptions to simple bad implementation choices that could be rewritten more efficiently.

Data integrity. As the current database was in danger of going down with the application at any moment, its integrity, as well as a safe backup strategy, were chief concerns in this project. On top of technical requirements, an easier way of handling and updating the database with sufficient privileges was considered a goal.

7 Migrating the Application

This chapter presents the steps of modernisation and migration plans based on the information gathered in the previous sections. It offers some suggested guidelines and implementation steps as to how to proceed in tackling the known technical problems while making the transition to a cloud environment smooth and useful. It then analyses the results compared to the goals set in the previous section to see what was achieved.

7.1 Codebase Update

As mentioned in the previous section, the project hadn't been updated in a while and was lagging behind in terms of code quality and framework compatibility. Before undertaking any major structural changes, the codebase would have to be updated and unified to make any further changes easier and more straightforward.

7.1.1 Goals

By undertaking a codebase update, the technical problems of the outdated codebase, the difficulty of adding new features, and the inferior performance of parts of the application would be targeted.

The goals of this step would be to flatten the learning curve, by having a better starting point into the application, as well as technically optimise the codebase, by removing low-quality code during the update. In addition, this step would increase such non-functional attributes as application security, by deleting unsafe code, performance, by optimising technical solutions, and maintainability, by having a uniform codebase.

7.1.2 Implementation

In the case of this project, updating the codebase meant a few main things.

Refactor the file structure. Since the application structure had fallen behind compared to the one recommended in the official guide, it was an important step to rearrange the project files into their intended folders. This would ensure their behaviour was consistent with that of the documentation, and make further development easier.

Structure file imports. Due to the functionality presented by the Meteor framework in its earlier versions, code was available everywhere in the project, whether this was intended or not. By restructuring the application file structure, all file and module imports would have to be checked as well to see that they were imported where needed, and ignored where not.

Take care of technical debt. As the framework had undergone some major changes, the code itself was also lagging behind in technical quality and implementation. Having consistent and high-quality code would benefit any further changes made to the application greatly, as well as optimise the current functionality and performance.

Enforce good development practices. Although having no effect on the application code written so far, enforcing good development practices would ensure that any further development done on the application would not add to the existing technical debt. By enforcing code reviews or making tests mandatory, the future quality of the codebase would be less likely to end up in a state such as it was at the beginning of this project.

7.1.3 Results

In practice, updating the codebase was a slow and laborious process, that took around two months to complete in a satisfactory manner. The result was a greatly improved codebase, which was valuable in itself, as well as a good prerequisite for any further development.

Refactoring the file structure and file imports was quite straightforward, as there were a clear guide and documentation to follow on Meteor's part. Both updates to structure and imports were implemented per file, as moving a file inside the project meant having to update and check the file imports as well. This step greatly improved the overall structure of the application and made it easier to go through the application code.

Taking care of technical debt was slightly trickier, as a few different things were involved. In this case, technical debt meant the quality of the code rather than other shortcomings, as well as the debt left by using old tools for development. In the end, the version of the Meteor framework was updated to 1.9 in the project, and while the code was made compatible with it, it didn't yet make use of the newer features introduced by this later version. This did, however, enforce having to delete no longer supported unsafe code from the application, which was one of the targeted goals. Most of the problems with the application speed were also explained by discovered bad technical implementations in the code, which were then remedied during this step.

While none of these changes were mandatory in regards to the migration, choosing not to implement them would have resulted in difficulties and longer throughput time later on in any modernisation step, as discussed in the section about code quality debt. Thus, updating the codebase was a crucial step of this project, and addressed many of the technical problems found in the application, such as the outdated codebase, and the difficulty of adding new features to the existing code. Also, some slower parts of the application benefited greatly from diminishing the overall technical debt. The results of enforcing good development practices will only be seen after the development of the application continues, and as such is not analysed here further.

7.2 Database Migration

One of the main problems and risks of the application was its database, and especially its location inside the application Docker container. To ensure the stability and maintainability of the application database, it would have to be migrated from its current location to a better alternative.

7.2.1 Goals

The main technical problem the database migration meant to solve was the instability of the current database implementation.

By successfully migrating the database to a safer environment, the goal of the application's data integrity would be reached. In terms of non-functional enhancements, this would mean better data integrity, as well as a strategy for backups and disaster recovery, which hadn't yet been implemented.

7.2.2 Implementation

In order to do ensure the database's stability, two things were required.

Database update. Since Meteor version 1.4, the default MongoDB version had been MongoDB 3.2, compared to the previous supported default version of 2.6 [21]. Some major updates were required between those versions, after which upgrading would be trivial. Although all MongoDB versions from 2.4 are still supported in Meteor, leaving the database to a deprecated version with little support elsewhere would hardly be a smart choice.

Database migration. As the current platform of the database was on unstable ground, the migration of its hosting platform was required. At the same time, setting up backups and database replicas would ensure data integrity and stability. For these purposes, the solution would be to move the database to a managed cloud database service, which would take care of the necessary additional requirements. Most managed database services only support relatively new versions of databases, so an update of the database would be a prerequisite of this step.

7.2.3 Results

The database migration was a surprisingly low-effort step, as there were many existing tools present to help with the migration. The update and migration work took around two weeks in total to implement, and the proposed benefits were immediately reached.

Updating the database was quite straightforward in this case, as the application hadn't been making use of any deprecated features of MongoDB in its database. MongoDB and Meteor both provided the necessary tools for running the update, as well as making a copy of the updated database for the upcoming migration.

The managed service MongoDB Atlas was chosen for hosting the application database. In addition to being a more stable environment than what was in use before, it offered additional services in terms of database replicas and automatic backups, which were a secondary goal of improving the overall data integrity. Pointing the existing application to the new database was then a simple matter of updating the application settings.

Updating and migrating the database addressed the problem of an unstable and unreliable database, as well as separated the responsibility of managing it to a separate service, leaving less work for the developers. It also added the beginning of a backup and disaster recovery strategy for the application.

7.3 Hosting Service Migration

As the application struggled with an uneven application load and difficulty in scaling, migrating the hosting platform and service seemed logical. This would comprise migrating the application onto a new hosting service provider, and finding a suitable managed service on top of which the application could safely run and scale as needed.

7.3.1 Goals

The main problems in focus during this step were the uneven server load, as well as the difficulty of applying updates to the application.

By changing the application hosting service, the goal of optimising the application infrastructure would be started. In addition, making it easier for maintainers to maintain and update the application would be achieved. All in all, better application availability, as well as improved scalability, would ensue.

7.3.2 Implementation

In order to change the hosting service of the application, a new environment was to be found, as well as a suitable service. After analysing the options, the following services were settled upon.

Google Cloud Platform. As the public cloud offered many services that would help with the goals of this optimisation, it was a logical choice to migrate the application hosting server onto a CSP. Since some of the services of Google Cloud Platform were already familiar, there would be a low learning curve for the platform itself, as well as the services it provided. Because of this, GCP was a logical choice for the application destination.

Google Kubernetes Engine. Since the application was already shipped in Docker containers, it seemed logical to use a container orchestration program such as Kubernetes to help with managing the operations part of hosting the application. Since GCP offers its Kubernetes as a Service GKE as part of its services, it was suitable as the target destination for the application containers.

7.3.3 Results

Migrating the application onto the public cloud was a lengthy process, and took the better part of three months. While small improvements to availability and maintainability were seen quite fast, the targeted goals required considerable additional effort in terms of getting to know and use the tools required for achieving the desired end result.

A lot of time was spent on learning about new services, as the chosen target destination platform, GKE, was not familiar beforehand. Changing the hosting platform also meant having to change the deployment tools used so far, as the previously used Meteor Up didn't have support for the kind of deployment strategy that would make use of GKE's desired benefits. This meant having to learn about Meteor's manual deployment tools, as well as getting them to work with GKE's update strategies.

In the end, GKE offered solutions to some of the technical problems related to updating and scaling the application. Automatically scaling the number of containers up or down and balancing traffic between them, depending on server load, served to reduce hosting costs and service shortages. A new update strategy, in this case rolling updates, ensured that the application wouldn't be completely unavailable at any time, but instead updated gradually, thus keeping its services online for its users, and increasing its overall availability.

7.4 Architecture Transformation

Since the application had grown considerably after its initial requirements, the designed architecture wasn't the best in terms of further development, or maintainability, and as such could be seen as having gathered some architectural debt. To be able to reach all of the proposed goals of the migration project, as well as achieve additional benefits, a redesigning and restructuring of the application architecture should be undertaken.

7.4.1 Goals

By rearchitecting the application, the technical problems of an uneven server load inside of the application, as well as the difficulty of adding new features, would be addressed.

This step would focus on the goal of optimising the underlying infrastructure, by dividing the main application into smaller pieces. This would also help in flattening the learning curve, since developers could focus on a smaller part of the application instead of having to familiarise themselves with it all.

This measure would further benefit the application's scalability, as well as its maintainability.

7.4.2 Implementation

In order to make the application architecture more suitable to its current use, as well as optimise it for adaptation to a cloud environment, the following steps are proposed.

Microservices architecture. To make full use of the benefits of the cloud, a microservices architecture would make the most sense. Separating application functionality into multiple services would make them more manageable and customisable, in terms of development, availability, and scaling. Having an automatically scaling container structure doesn't benefit the application greatly if the application load is uneven inside of the container.

Managed services. Besides separating the application into microservices, some parts of the application could be migrated away from the core application, and make use of separately managed cloud services, e.g., for user authentication. Not all parts of the application are migratable to such services, however, or if they are, the migration work might end up being significantly higher than the benefits it would bring.

7.4.3 Results

Redesigning and transforming the application architecture was seen as such a major undertaking that it was decided to be implemented as a partial rewrite at a later date. Instead of designing and transforming the whole application structure at this time, it would be implemented little by little, in parallel to developing and detaching possible new features. As it stood, the current application architecture was so entwined that it was impossible to separate any functionality from it into a separate service without undertaking a considerable amount of work.

Moving the monolithic application to the cloud, as was implemented in the previous step, was not enough to make it as cost-effective and scalable as was targeted. Optimising the infrastructure through rearchitecting the application would tackle problems with the application load and related costs. As mentioned earlier, scaling the whole application up and down is little better than the previous vertical scaling tactic. Instead, scaling only parts of the application under varying traffic would be much more efficient, cost-wise and in regard to availability.

Having a well-structured microservices architecture would also tackle the difficulty of adding new features to the project. Since only the application associated with the wanted new feature would need to be familiarised with before taking on the development process, developers would have a much lower threshold when undertaking such a task.

All in all, even though a microservices architecture would make the most sense in a cloud-based environment, some benefits were reaped even without a lengthy and laborious rearchitecting process, but not nearly all that the new cloud environment could bring. The compromise of the implemented approach was to make the migration work with the current application as it was, and gradually work to separate some functionality into separate services, slowly collecting the benefits of its new environment.

8 Conclusions

This chapter presents the findings of this thesis and reflects on their applicability to other modernisation and migration projects. It answers the research questions presented at the beginning of this thesis based on the knowledge gained throughout this process.

8.1 Prerequisites of Migration Projects

The first research question was: *"What are the prerequisites of migrating a web application to the cloud?"*. In the research on technical debt, we saw the effect that badly maintained and low-quality code had on any project, regardless of its development methodology or current state in its life cycle. In the case study project, it was concluded that before undertaking any larger work, it would be greatly beneficial to have a codebase in good shape and up to date with its tools, such as its framework, as well as free of any outstanding technical debt that might hinder further development. The use of modern application development approaches, such as TDD and CI/CD, would be a great help in ensuring a low level of technical debt from the start of any software development project.

Another factor that was seen to impact the success of migration projects is the suitability of the current application architecture to its planned destination platform. If a monolithic application was to be migrated to a single VM instance, no additional measures would be required in order to successfully complete said migration. If a microservices application or a part of it would be migrated onto a cloud platform, the migration should be quite straightforward, assuming that corresponding services are available. In the case study application, we saw that while the migration of a larger monolithic application to a cloud environment is possible, the outcome and achieved results might vary from those that were hoped, depending on what the desired benefits of the migration were in the first place.

8.2 Modern Technology Considerations

The second research question was: *"What needs to be taken into account when dealing with modern technologies?"*. Here again, we find that technical debt has an important part to play in the answer. In the research on modern technologies, we concluded that projects implemented with modern technologies have a higher risk of accumulating technical debt than ones done with more established technologies, although the technologies themselves might then harbour more debt. As we concluded in our research on technical debt, special attention should always be paid to ensure that debt is kept at a manageable level, as it hinders all future development.

Another thing to consider, when working with modern technologies, is that they might encounter a sudden end of life. This needs to be taken into account at the very beginning of a project and accounted for in its life cycle plan. In the case of a sudden end of life, a modernisation process of migrating the technology, rather than the platform, might be undertaken.

8.3 Hosting on the Cloud

The third and final research question was: *"What benefits and drawbacks are there in hosting a web application on the cloud?"*. In the research on the public cloud and CSPs, we saw that among others, the public cloud offered a variety of benefits, especially when it came to non-functional qualities, such as availability, security, and scalability. While some of the risks and drawbacks, including access limitations and strict security concerns, are still present in the public cloud, in the scale of this case study, the concerns weren't great enough to outweigh the possible benefits.

One major drawback that was encountered during the case study was the amount of time spent on learning new services, as the learning curve turned out to be quite steep. When planning a timeline for a migration project, the time to learn must be taken into account to avoid surprising delays in implementation. However, if the services happen to be familiar to the developers beforehand, no such learning time is needed.

While a public cloud environment does offer various benefits, they are not automatically attained once an application has been migrated to a public cloud environment. On the contrary, if done hastily and on the wrong services, a faulty migration might cause more drawbacks than benefits, e.g., by accumulating costs on services that haven't been optimised or tailored to fit the application.

In this case, despite the workload and risks involved in migrating a web application to the public cloud, the case study showed that some advantages might be gained from a migration to the cloud, despite not having tailored the application to the targeted destination. However, as such a task still requires a considerable amount of work, it should be undertaken only after careful consideration, e.g., as a step before further optimising an application to fit a cloud environment.

References

[1] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. *Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud.* In 2015 10th Computing Colombian Conference (10CCC) (pp. 583-590). IEEE.

[2] Hall W., Tiropanis, T. *Web Evolution and Web Science.* Computer Networks. 2012

[3] Ravula, S. *Achieving Continuous Delivery of Immutable Containerized Microservices with Mesos/Marathon.* 2017

[4] *Microservices Decoded: Best Practices and Stacks.* DZone. `https://dzone.com/articles/scalable-cloud-computing-with-microservices`. Accessed 16.5.2020.

[5] Miri, I. *Microservices vs. SOA.* 2017 `https://dzone.com/articles/microservices-vs-soa-2`. Accessed 29.2.2020.

[6] Fowler, M. *Microservices - a definition of this new architectural term.* 2014. `https://martinfowler.com/articles/microservices.html`. Accessed 29.2.2020.

[7] Bogner, J., Zimmermann, A., & Wagner, S. *Analyzing the Relevance of SOA Patterns for Microservice-Based Systems.* ZEUS, 9, 9-16. 2018.

[8] Mesbah, A., van Deursen, A. *Migrating Multi-page Web Applications to Single-page Ajax Interfaces.* 2006

[9] Neoteric. *Single-page application vs. multiple-page application.* 2016 `https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58`. Accessed 1.3.2020.

[10] Yan, K. *Modern web application frameworks.* Accessible at `https://bravoka.io/articles/modern-web-application-frameworks/`. Accessed 28.1.2020.

[11] Majchrzak, T. A., Biørn-Hansen, A., & Grønli, T. M. *Progressive web apps: the definite approach to cross-platform development?.* 2018.

[12] Offutt, J. *Quality attributes of web software applications.* IEEE software, 19(2), 25-32. 2002.

[13] George, B., & Williams, L. *A structured experiment of test-driven development.* Information and software Technology, 46(5), 337-342. 2004.

[14] Fowler, M. *Continuous integration.* 2006. `http://martinfowler.com/articles/continuousIntegration.html`. Accessed 27.2.2020.

[15] Fowler, M. *Continuous delivery.* 2013. https://martinfowler.com/bliki/ContinuousDelivery.html. Accessed 27.2.2020.

[16] Hukkanen, L. *Adopting Continuous Integration – A Case Study.* 2015.

[17] *GitHub - angular/angularjs: AngularJS - HTML enhanced for web apps!.* GitHub. https://github.com/angular/angular.js. Accessed 16.5.2020.

[18] Strack, I. *Getting Started with Meteor.js JavaScript Framework.* Packt Publishing Ltd. 2015.

[19] Rust, S., Schelling, J., & Schipper, D. *Building Real-Time Web Applications with Meteor.* 2015.

[20] *Contributors to meteor/meteor.* GitHub. https://github.com/meteor/meteor/graphs/contributors. Accessed 16.5.2020.

[21] *Meteor Changelog.* Meteor API Docs. https://docs.meteor.com/changelog.html. Accessed 11.5.2020.

[22] *The State of JavaScript 2019: Meteor.* The State of JavaScript 2019. https://2019.stateofjs.com/back-end/meteor/. Accessed 16.5.2020.

[23] *Application Structure.* Meteor Guide. https://guide.meteor.com/structure.html. Accessed 9.5.2020.

[24] *Deployment and Monitoring.* Meteor Guide. https://guide.meteor.com/deployment.html. Accessed 1.4.2020.

[25] *Scalable Hosting with Galaxy for Meteor Applications.* Meteor. https://www.meteor.com/hosting#pricing. Accessed 1.4.2020.

[26] Leau, Y. B., Loo, W. K., Tham, W. Y., & Tan, S. F. *Software development life cycle AGILE vs traditional approaches.* In International Conference on Information and Network Technology (Vol. 37, No. 1, pp. 162-167). Product-Focused Software Process Improvement, 26. 2012.

[27] Yli-Huumo, J., Maglyas, A., & Smolander, K. *Evaluating and managing technical debt in software development lifecycle.* Product-Focused Software Process Improvement, 26. 2014.

[28] Kruchten, P., Nord, R. L., & Ozkaya, I. *Technical debt: From metaphor to theory and practice.* Ieee software, 29(6), 18-21. 2012.

[29] Raksi, M. *Modernizing web application: case study.* 2017.

[30] Tibus. *The history of web hosting: how things have changed since Tibus started in 1996.* https://www.tibus.com/blog/the-history-of-web-hosting-how-things-have-changed-since-tibus-started-in-1996/. Accessed: 1.2.2020

[31] Malhotra L, Agarwal D, Jaiswal A. *Virtualization in cloud computing.* J. Inform. Tech. Softw. Eng. 2014 Jun;4(2):1-3.

[32] Xavier MG, Neves MV, Rossi FD, Ferreto TC, Lange T, De Rose CA. *Performance evaluation of container-based virtualization for high performance computing environments.*. In2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing 2013 Feb 27 (pp. 233-240). IEEE.

[33] Molnar, D., Schechter, S. *Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud.* WEIS, 2010.

[34] Kamal, M. A., Raza, H. W., Alam, M. M., & Su'ud, M. M. *Highlight the Features of AWS, GCP and Microsoft Azure that Have an Impact when Choosing a Cloud Service Provider.* International Journal of Recent Technology and Engineering (IJRTE). 2020.

[35] Brandao, P. R. *Computer Forensics in Cloud Computing Systems.* Budapest International Research in Exact Sciences, 1(1), 02. 2019.

[36] Youseff, L., Butrico, M., & Da Silva, D. *Toward a unified ontology of cloud computing.* In 2008 Grid Computing Environments Workshop (pp. 1-10). IEEE. 2008.

[37] *UpCloud - High Performance Cloud Hosting.* `https://upcloud.com/`. Accessed 6.4.2020.