A Case Study on Cloud Migration and Improvement of Twelve-Factor App

Master of Science Thesis
University of Turku
Department of Computing
Software engineering
2021
Miika Peltotalo

UNIVERSITY OF TURKU
Department of Computing

Miika Peltotalo: A Case Study on Cloud Migration and Improvement of Twelve-Factor App

Master of Science Thesis, 52 p.
Software engineering
June 2021

The Twelve-Factor app methodology was introduced in 2011, intending to raise awareness, provide shared vocabulary and offer broad conceptual solutions. In this thesis, a case study was done on two software implementations of the same business idea. The implementations were introduced and then analyzed with Twelve-Factor. Hevner's Information Systems Research Framework was used to assess the implementations, and Twelve-Factor's theoretical methodology was combined with them to form results.
The implementations were found to fulfill most of the twelve factors, although in different ways. The use of containers in the new implementation explained most of the differences. Some factors were also revealed to be standard practices today, which showed the need to abstract factors like Dependencies, Processes, Port binding, Concurrency, and Disposability.
In addition, the methodology itself was analyzed, and additions to it were introduced, conforming to the modern needs of applications that most often run containerized on cloud platforms. New additions are API First, Telemetry, Security, and Automation. API First instructs developers to prioritize building the APIs at the start of the development cycle, while Telemetry points that as much information as possible should be collected from the app to improve performance and help to solve bugs. Security introduces two different practical solutions and a guideline of following the principle of least privilege, and lastly, automation is emphasized to free up developer time.

Keywords: Twelve, factor, app, methodology, architecture, AWS, best practices

Contents

List of Figures

List of Tables

List of acronyms

API	Application Programming Interface
APM	Application Performance Monitoring
AWS	Amazon Web Services
CD	Continuous Delivery
CDN	Content Delivery Network
CI	Continuous Integration
CSS	Cascading Style Sheets
DDoS	Denial-of-service attack
DevOps	Software development and IT operations
DNS	Domain Name System
DOM	Document Object Model
EC2	Elastic Compute Cloud
ECR	Elastic Container Registry
HCL	Hashicorp Configuration Language
HTML	HyperText Markup Language

HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
JAR	Java ARchive
JSON	JavaSript Object Notation
JSX	JavaScript XML
JVM	Java Virtual Machine
JWT	JSON Web Tokens
NPM	Node Package Manager
NVM	Node Version Manager
ORDBMS	Object-Relational Database Management System
ORM	Object–relational mapping
PaaS	Platform as a Service
RDS	Relational Database Service
REST	Representational state transfer
RxJS	Reactive Extensions for JavaScript
S3	Simple Storage Service
SG	Security Group
SLF4J	Simple Logging Facade for Java
TLS	Transport Layer Security

UI	User Interface
UX	User Experience
VM	Virtual Machine
VPC	Virtual Private Cloud

1 Introduction

1.1 Scope and Goal

There are so many ways to build software and not many guides that describe different parts and requirements of the whole software product. This thesis describes two different software products and their architectures while also comparing them to the Twelve-Factor App methodology.[1] The goal is to determine if the new implementation fulfills a set of best practices introduced by Twelve-Factor better than the old implementation and whether these best practices are still up to date. Kevin Hoffman's Beyond the Twelve-Factor App[2] is also used as a reference to achieve this.

This thesis focuses on architecture and how the data flows through the application rather than a single technology. Some languages and frameworks are covered very briefly, and there will also be examples of them, but the main idea is to show the parts of the software product and how Twelve-Factor describes them.

Most of the available literature on software architecture is often specific to backend implementations, such as microservices. There are not many papers on full-stack implementations that also include Infrastructure as a Service solution, providing a complete overview of the software product. This thesis aims to fill that gap by providing a complete overview of the implementations and proposing best practices for product development.

1.2 Problem Description

In this thesis, we are going to describe two solutions to an existing customer problem. The first is the original implementation, while the second is the improved rewritten cloud application. Both will be analyzed with the Twelve-factor app methodology as a base.

The main idea is to find how well these implementations follow best practices described in the Twelve-Factor App and whether there are parts that are not covered by this methodology. The methodology was published in 2011 by Adam Wiggins, and because of its age, there is a high possibility that there can be modern additions to it.

1.3 Research methods

As a research framework, Hevner et al. Information Systems Research Framework[3] is used. As described in Figure 1.1 we are using Methodologies, Twelve-Factor app, to evaluate two different code bases as a case study. These codebases are studied as artifacts.

As a result, we can assess our study methods and whether there is something that we can add to the tools we are using to evaluate code bases. The framework also considers the environment in which we are working because the stakeholders always affect possible outcomes with the business needs.

In Figure 1.2 Håkansson describes a portal of research methods and methodologies.[4] Its primary purpose is to support our selection of research methods. Håkansson's portal shows methods that we can use to gather data about the environment and knowledge base so that we can use them, just like in Figure 1.1, applying business needs from the environment and practical knowledge from the knowledge base.

Because this thesis investigates real practical work, the research method is ap-

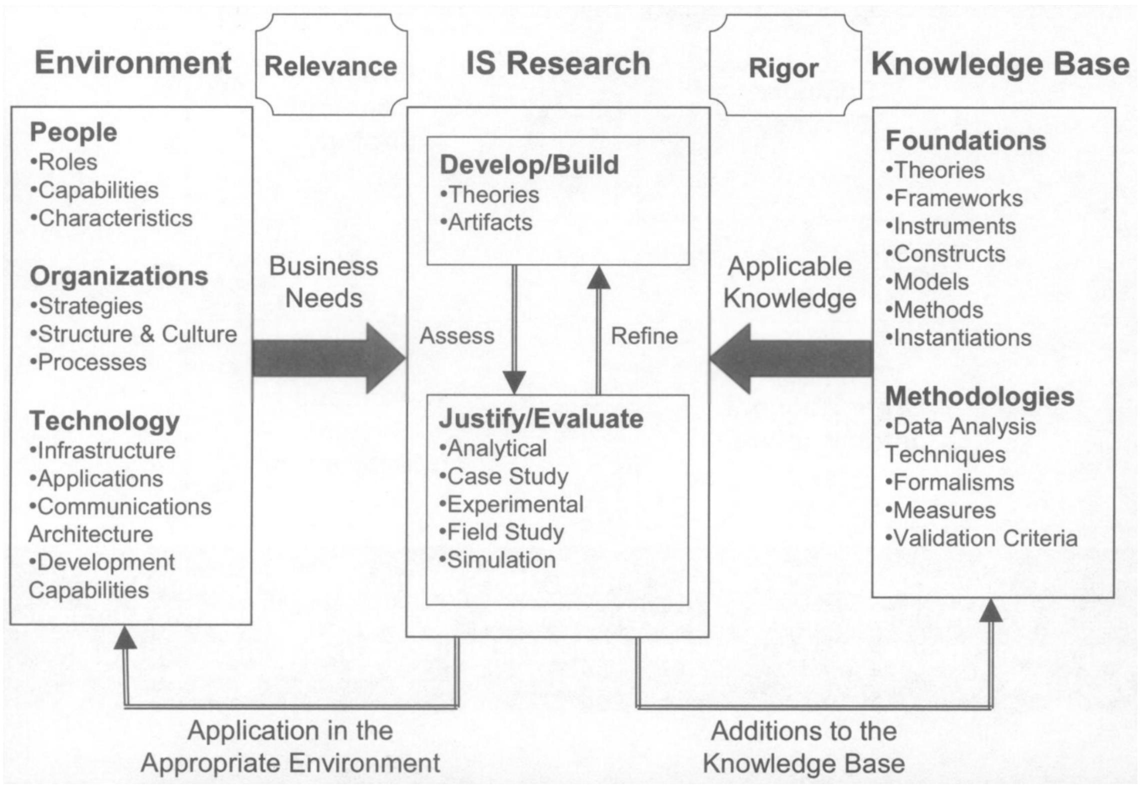

Figure 1.1: Information Systems Research Framework [3]

plied research[4] which is used in works that have actual implementations. From the research approaches, abductive approach[4] is in use, which derives likely conclusions from an incomplete set of observations identified from the two different software implementations. These implementations act as cases from which the analysis is derived, and improvements for the practical work are suggested.

We are using a case study for research strategy and data collection from the portal of research methods, an empirical study investigating a phenomenon in a real-life context where boundaries between phenomenon and context are not evident.[4] It also involves an empirical investigation using multiple sources of evidence. In this case study, we are analyzing the old implementation and its replacement candidate. They both are real-world applications, the latter having been built just before this thesis was written. The aim is to find from this analysis how applications are built and how valid Twelve-Factor is as a guideline.

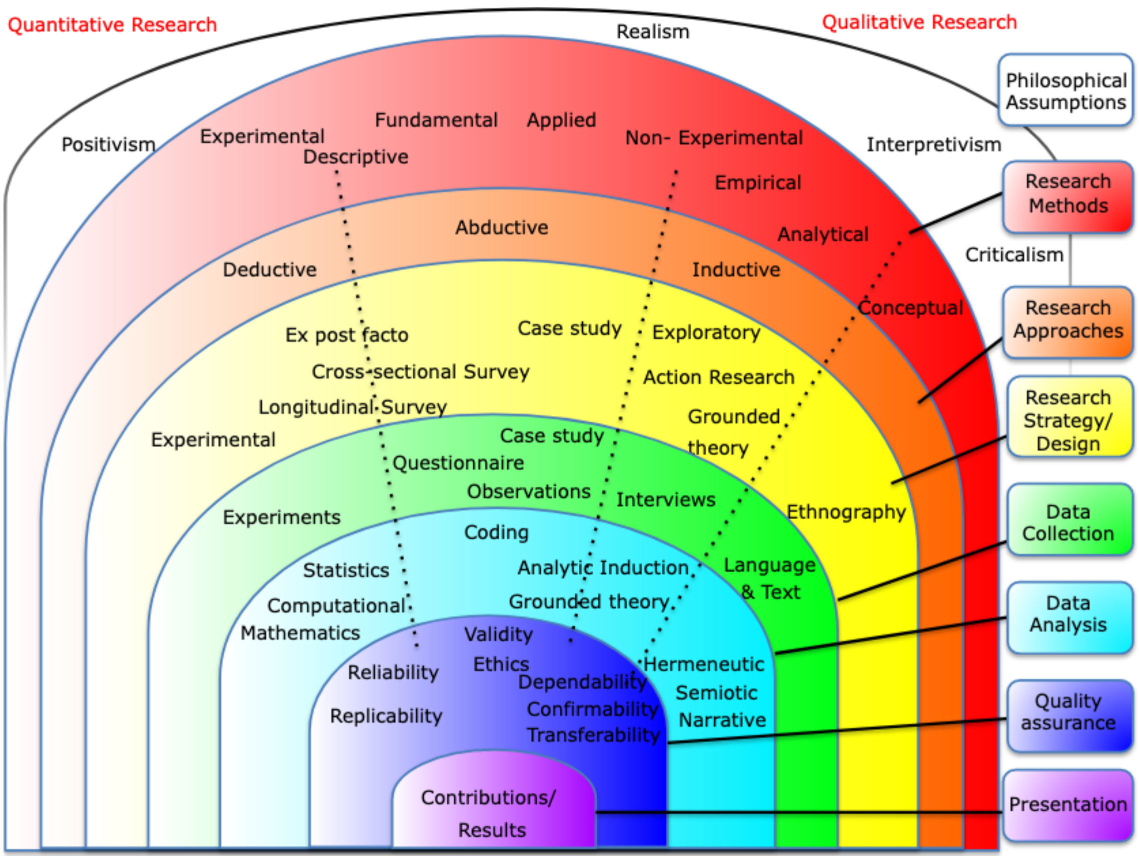

Figure 1.2: Portal of Research methods [4]

The original application and the new implementation provide us with data about our business needs in the thesis. Twelve-Factor App methodology works as ground truth, providing practical knowledge for our research. These are used to conclude whether the old and new implementations match the Twelve-Factor steps. Also, we will look at Twelve-Factor critically and find out how valid it is as ground truth.

1.4 Stakeholders and Delimitation

This work has been done for Identio Oy. Identio builds technology services and provides UI/UX expertise. An improved application described in Chapter 4 has been built for a client during employment at Identio. No client information is disclosed in this thesis, and the general architecture could be applied to any other software product.

At the initial stage, the client had a couple of wishes. The first was that backend technology should be JVM-based and that the AWS migration from their current server should be continued. These were the only requirements set at the start of the project. The current technical stage of the product is a proof-of-concept, and the status has been frozen for this work.

1.5 Outline

In Chapter 1, I introduce a general description of the problem and the research methods used. Next, in Chapter 2 I explain the technologies used in both implementations of the product. Continuing to Chapter 3 I describe the previous implementation, which is then replaced by the newer implementation. I also analyze it using the Twelve-Factor app methodology. In Chapter 4 I introduce the replacement candidate and also analyze it with Twelve-Factor. The second last Chapter 5 is where I synthesize what is learned from both of these Twelve-Factor analysis steps and introduce the new Ten-Factor model. In Chapter 6 I draw my conclusions and set up the basis for future research.

2 Technical Background

We will describe central technologies used in two of our implementations in the technical background while also describing concepts that the Twelve-Factor App methodology is based on. In both implementations, the web development technologies provided tools to present and transfer data, then saved through different operations to the persistence layer.

The development process was made more straightforward by using virtualization, version control, build tools, and DevOps. These tools are essential for the development process, and some of the themes also appear in Twelve-Factor. Later, Cloud services, Microservices, and the Twelve-Factor App methodology itself are introduced. While Cloud services describe different cloud strategies and platforms used in our new implementation, Microservices and Twelve-Factor then describe the actual patterns and broad concepts.

2.1 Web development

Web development most often refers to the Client-Server model where the client is the frontend of the application, and the server is the backend. Because of technological advancements in browsers during recent years, it is common to implement even large applications that run entirely in the browser.

Traditional web applications run in the browser and are built with HTML, CSS, and JavaScript. The applications most often render data supplied by the backend over HTTP requests. HTML elements form the Document Object Model, which is styled using CSS and the data is manipulated and inserted into document object model(DOM) using JavaScript.

2.1.1 JavaScript and TypeScript

JavaScript is a multi-paradigm dynamic language with types and operators, standard built-in objects, and methods.[5] TypeScript then builds on JavaScript by adding static type definitions to the language. Types can provide a way to describe objects, which provides better documentation and allows TypeScript to validate objects during compilation.[6]

Listings 1 and 2 show the difference between JavaScript and TypeScript. JavaScript is more concise compared to TypesScript, but TypeScript adds additional type safeguards to primitive types while developers can also define interfaces for objects.

Listing 1 Example of JavaScript.

```
class UserAccount {
    constructor(name, id) {
        this.name = name;
        this.id = id;
    }
}
const user = new UserAccount("Murphy", 1);
```

In the frontend, JavaScript or TypeScript is used to manipulate and insert data into DOM, creating the basic UI of the web app. They can also be used in the backend to build lightweight REST APIs. The backend in this thesis is powered by a different language than what is found at the frontend.

Listing 2 Example of TypeScript.

```
interface User {
  name: string;
  id: number;
}

class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}
const user: User = new UserAccount("Murphy", 1);
```

2.1.2 React

React is a declarative JavaScript library for building user interfaces. The main idea of React is that the developer can compose larger pieces of software from smaller isolated pieces of code, which React calls Components.[7] With React you can describe the UI with an HTML-in-JavaScript syntax called JSX. However, it is just syntactic sugar on top of React, and the actual JSX translates into React.createElement() function.[8]

Unlike many frameworks, React does not enforce code conventions or file organization[9]. These are left to teams and individuals to Figure out best practices. Also, because React is not an actual framework, it needs supporting libraries for routing

Listing 3 Example of React component.

```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

and DOM manipulation. Having much freedom can be a good thing, especially when teams have a clear idea about the project structure, but this can also backfire if conventions are not discussed.

2.1.3 Angular

Unlike React, Angular is a more comprehensive solution that aims to provide everything needed for frontend development. Angular features are a component-based framework, a collection of integrated libraries including routing, forms management, client-server communication, and a suite of developer tools to cover each part of the development cycle.[10]

Listing 4 describes a simple Angular component. Angular uses decorators to inject configuration metadata for components, which instructs Angular how the components should be processed. Here a TypeScript class HelloWorldComponent is given a decorator @Component.

2.1.4 Kotlin and Ktor

Kotlin is marketed as a concise, safe, and interoperable language. It uses Java Virtual Machine(JVM) as its base, and it has been built to be compatible with

Listing 4 Example of Angular component.

```
import { Component } from '@angular/core';


@Component({
  selector: 'hello-world',
  template: `
    <h2>Hello World</h2>
    <p>This is my first component!</p>
    `,
})
export class HelloWorldComponent {
  // The code in this class drives the component's behavior.
}
```

Java libraries.[11] Other benefits of Kotlin are its conciseness and null safety, which otherwise would lead to a situation where accessing a member of a null reference will result in a null reference exception[12]. Since Kotlin objects already have essential functions like getters and setters built-in, the developer does not need annotation preprocessor libraries like Lombok in Java to include getters and setters in their classes.[13]

Ktor is built by Jetbrains, the same company that made Kotlin, and its purpose is to be a lightweight asynchronous framework for building microservices. In this context, it is used as a REST service that supplies data to the frontend. Ktor was chosen as a backend technology for its low overhead approach.

The main principles of Ktor are listed as unopinionated, asynchronous, and testable. It does not impose many constraints on what tools or technologies should be used and instead allows users to choose themselves. Being a Kotlin framework, the asynchronous coroutines allow developers to avoid thread blocking.

2.2 Data persistence with PostgreSQL

PostgreSQL is an open-source object-relational database management system (ORDBMS). It follows the SQL standard very closely, supporting 160 out of 179 features required for full core SQL:2011 compliance. PostgreSQL also has the open-source community behind it, and it was the second most popular database technology in Stack Overflow Developer Survey 2020[1].

2.3 Virtualization

Operating system-level virtualization means partitioning the operating system into isolated virtual machines (VM) that can be used as a base for software services. These VMs have small runtime and little resource overhead with better security since the underlying operating system cannot be accessed. These capabilities make them ideal to be used in a service platform.[14]

Docker is one of these operating system-level virtualization platform solutions that can be used to deliver software. It offers isolated modules called containers with their operating system kernel and bundled software with configuration files. Docker solves the works-on-my-machine problem developers struggle with when having multiple environments where the software is supposed to run by offering a standard unit that houses software. Each container can be run in any environment as long as it has a Docker Engine.[15]

[1]`https://insights.stackoverflow.com/survey/2020#technology-most-loved%2Ddreaded%2Dand-wanted-databases`

Listing 5 Example of Dockerfile.

```
FROM ubuntu:18.04

COPY . /app

RUN make /app

CMD python /app/app.py
```

2.4 Version control

The purpose of version control is to record changes to a file or a set of files so that the developer can recall specific versions later. With version control, it is possible to revert selected files to a previous state, revert the entire project to a previous state, compare changes over time or pinpoint where the breaking changes happened in the project.[16]

The most commonly used version control system today is Git, an open-source distributed version control system. Few service providers host Git: Github, GitLab, Bitbucket, AWS Codecommit, and many others. According to Stackoverflow 2020 developer survey[2], most developers use Github or GitLab as their version control hosting service.

2.5 Build tools

Build tools are solutions that package and deliver raw source code into running applications. This step usually involves compiling, optimizing, testing, and packaging. Automatization of these steps helps boost productivity since more time is freed for actual development. Build tools can also manage dependencies or assets of the application.

Gradle is an open-source build automation tool for JVM environments like Maven

[2]https://insights.stackoverflow.com/survey/2020#technology-collaboration-tools

or Apache Ant. It is used to run tasks and manage dependencies, and it also comes with its plugin system that allows using ready-made implementations for common use cases. These tasks are then used to compile .jar files, run tests or create documentation.[3]

Listing 6 Example of build.gradle file.

```
group 'Example'
version '1.0-SNAPSHOT'
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.4.10"
  }
}
apply plugin: 'kotlin'
repositories {
  jcenter()
}
dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.4.10"
  testCompile group: 'junit', name: 'junit', version: '4.12'
}
```

Webpack is a static module bundler for JavaScript that processes and builds a dependency graph of the application. Based on this, it generates one or more bundles. Without configuring, it understands JavaScript and JSON, but with its

[3]https://docs.gradle.org/current/userguide/what_is_gradle.html#what_is_gradle

in-built loaders, it is possible to process other types of files. Webpack can be further extended with plugins used for bundle optimization, asset management, and injecting environment variables.

Its idea is that every file in the project is a module. In bundling, process modules are then combined into chunks which are further combined into chunk groups that form the so-called ChunkGraph. Chunks can be of two types: Initial chunk, the main chunk of the entry point, and Non-initial chunk that may be lazy-loaded. Initial chunks contain modules, their dependencies specified for the entry point, and non-initial chunks that can appear during dynamic imports or when using specific plugins.[4]

Listing 7 Example of webpack.config.js file.

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  }
};
```

2.6 DevOps

DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services.[17] It is about removing obstacles between development and operations while delivering software continuously and with a shorter feedback loop. Continuous delivery and a shorter

[4]`https://webpack.js.org/concepts/under-the-hood/`

feedback loop can be achieved by following a set of best practices like automating build pipelines, using microservices, having monitoring and logging, automating infrastructure, and emphasizing communication and collaboration.

2.6.1 CI/CD

Continuous Integration(CI) and Continuous Deployment or sometimes Delivery(CD) are fundamental concepts of modern-day DevOps. While software projects are growing larger, it is important to automate repetitive work to free time for the actual development. CI/CD pipelines are an answer to this problem.

CI's basic idea is that the software is built, tested, and merged to a repository when changes happen. It ensures that problems are found frequently and makes them easier to fix in committed chunks. One example of a tool used for automated builds and tests is Gradle, introduced earlier.

CD can be split into two parts: Continuous delivery and continuous deployment. Continuous delivery is a step after CI releasing the validated code to a repository and creating a release branch. Continuous deployment then follows this by delivering the contents of that branch into production. In web development, this usually means making the application available to users through cloud hosting.

2.6.2 GitHub Actions

GitHub Actions is a tool provided by GitHub that can be used to create workflows and triggers in code repositories. Actions can be configured to listen to specific events like code being pushed to the repository. These events then trigger workflows that can be configured to deploy the application to the production environment.

Actions can be configured using a human-readable data-serialization language called YAML. YAML files are stored in the repository and contain whole CI/CD pipelines that trigger Git commands or trigger new builds when external data changes. Below is an example of a simple workflow that checks the dependency version when new code is pushed into the repository.

Listing 8 Example of GitHub Actions.

```
name: learn-github-actions

on: [push]

jobs:

  check-bats-version:

    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v2

      - uses: actions/setup-node@v1

      - run: npm install -g bats

      - run: bats -v
```

2.7 Cloud services

Cloud services often refer to infrastructure, platform, or software that third-party service provider hosts. They abstract and pool scalable resources that are shared over a network. These services facilitate apps or software services and make them available to users. Cloud computing platforms then take this abstraction further, providing access to virtual computing environments with a single click.[18]

When it comes to choosing how a developer wants to host applications, there are generally two main choices: Infrastructure as a Service(IaaS) or Platform as a Service(PaaS). The difference between them is that PaaS offers application development and deployment platform, facilitating development and deployment services without the extra resources needed for setting up underlying services.[19]

Together with Google Cloud and Microsoft Azure, AWS is one of the most popular options for a cloud hosting platform. AWS offers flexible pricing and an extensive catalog of service options. Actual servers are distributed around the world and are then categorized as "hosted zones" in AWS. One problem, however, is that not every AWS service has been implemented for all hosted zones by the provider. For example, see in Chapter 4 Cloudfront is only hosted in North America.[20]

Terraform is an open-source infrastructure as code(IaC) software tool that can be used to configure and manage cloud services. It provides a command-line interface that can be used to pull and push the state into the chosen cloud service provider. This state is then applied to provision chosen resources in the cloud service.[21]

2.8 Microservices

Microservices architecture structures the application as a set of loosely coupled, collaborating services.[22] The opposite of microservices is a monolith application which is a single, autonomous unit. In Figure 2.1 architecture difference between the monolithic applications and microservices is shown. Monolith architecture on the left shows that all services are inside one application that communicates with its database. On the right is microservices architecture which separates the services into application modules and even databases.

A monolithic application becomes a microservice architecture when individual services are separated into modules. This separation can be done following Robert C. Martin's Single Responsibility Principle[23], a well-known programming principle that promotes clear responsibilities between modules. These services then communicate over application programming interfaces, having a clear contract on what individual service provides.

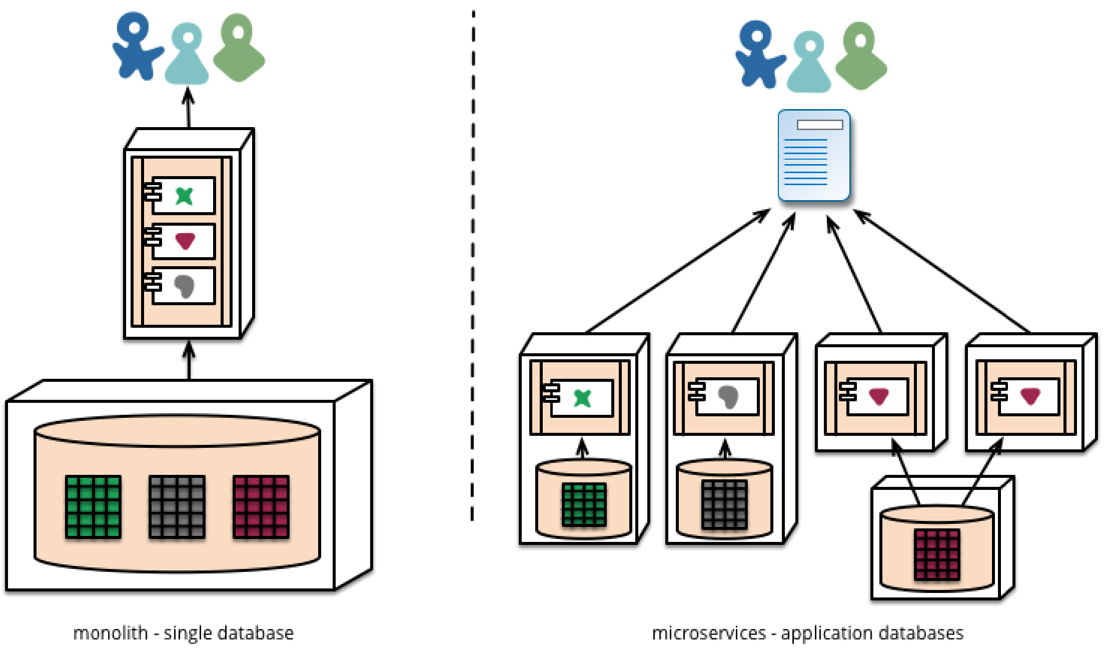
monolith - single database
microservices - application databases

Figure 2.1: Decentralised data

2.9 The Twelve-Factor app methodology

Twelve-Factor app methodology emerged in 2011 intending to raise awareness, provide shared vocabulary and offer broad conceptual solutions.[1] It was drafted by developers at Heroku[5] and written by Adam Wiggins. The methodology has been criticized in their competitors blog[24] for not having a general microservices approach and being more platform-specific to Heroku's platform service.

Twelve-Factor app has five main characteristics[1]:

- Use declarative formats for setup automation to minimize time and cost for new developers joining the project.
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments.
- It is suitable for deployment on modern cloud platforms, obviating the need

[5]`https://www.heroku.com/`

for servers and systems administration.

- Minimize divergence between development and production, enabling continuous deployment for maximum agility.
- It can scale up without significant changes to tooling, architecture, or development practices.

Ultimately, the goal is to have a guideline that makes it easier to work with the growing codebase, promote good ways of working between developers, and avoiding deterioration of software quality over time.

The twelve factors described in the methodology are as follows:

1. Codebase
    - One codebase per service, tracked in revision control, many deploys.
2. Dependencies
    - Explicitly declare and isolate dependencies.
3. Config
    - Store config in the environment.
4. Backing Services
    - Treat backing services as attached resources.
5. Build, Release, Run
    - Strictly separate build and run stages.
6. Processes
    - Execute the app as one or more stateless processes.

7. Port Binding
   - Export services via port binding.
8. Concurrency
   - Scale out via the process model.
9. Disposability
   - Maximize robustness with fast startup and graceful shutdown.
10. Dev/Prod Parity
    - Keep development, staging, and production as similar as possible.
11. Logs
    - Treat logs as event streams.
12. Admin Processes
    - Run admin/management tasks as one-off processes.

The first factor recommends one codebase per app as shown in Figure 2.2 where deploys are then made from that codebase. From a microservices point of view, the correct approach is one codebase per service.[24] These services are then deployed separately, deployed meaning a running instance.

The second factor declares that twelve-factor apps never rely on the implicit existence of system-wide packages. All dependencies are declared via a dependency declaration manifest both in development and production. Dependency isolation tool is used during execution to ensure no implicit dependencies leak in.[1]

The third factor explains that the app's config covers everything likely to vary between deploys of the codebase, for example, credentials and hostnames. The point

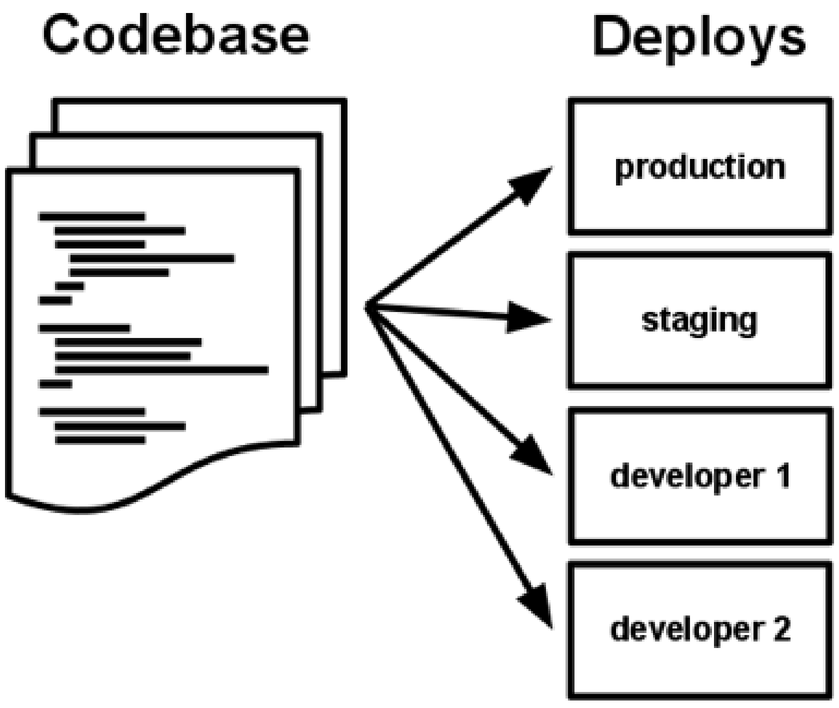

Figure 2.2: Codebase deploys

is to avoid storing any constants in the code because twelve-factor apps require strict separation of config from code.[1]

The fourth factor explains that each backing service should be treated as an attached resource. Just like in section 2.8 resource modules like datastores are consumed over the network. In the code, no distinction is made between local and third-party services.[1] This ensures maximum portability and flexibility since making changes to one module does not require building the whole application. In Figure 2.3 can be seen how production deploy combines each attached resource and communicates with them over the network.

To achieve the fifth factor, build, release and run, the codebase should be transformed into a complete release through the corresponding three steps, just like in Figure 2.4 first building the codebase and bundling dependencies into a single executable.

Then in the release stage, combining config with build to make a complete release

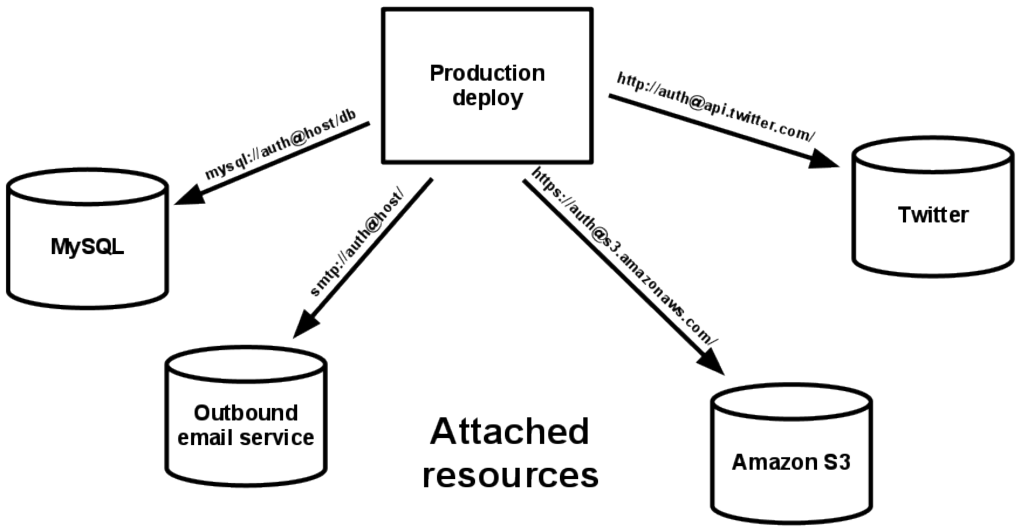

Figure 2.3: Backing services / Attached resources

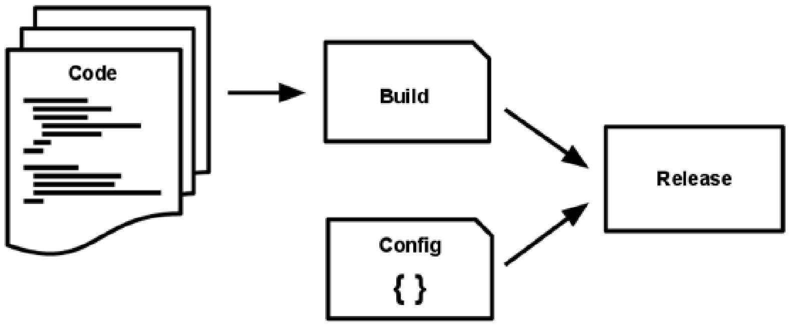

Figure 2.4: Build, release, run

and finally running the released product. CI/CD is recommended to automate the build, and with Docker images, it is effortless to separate the build and run stages.

The sixth factor focuses on app processes being stateless and share-nothing principled. Any persisting data should be stored in a stateful backing service. Memory space or filesystem is still allowed to be used as a single-transaction cache.[1] This makes scaling easier since adding more services is all that is needed for scaling horizontally.

The seventh factor means that the web app should export HTTP as a service by binding to a port and listening to requests coming in on that port.[1] Previously, when port binding was not as popular, the webserver was injected into the execution environment. The modern twelve-factor app should therefore be completely self-contained.

In eight factor processes are promoted as first-class citizens. Scaling out should be done relying on the underlying operating system, although individual processes can still handle internal multiplexing. In Figure 2.5 scaling out is shown. In horizontal axis is different process types that the app needs, and then in the vertical axis is scale shown by stacking process types. A web process handles HTTP requests, and worker processes are handling long-running background tasks.[1] This factor can easily be achieved using containerized services.

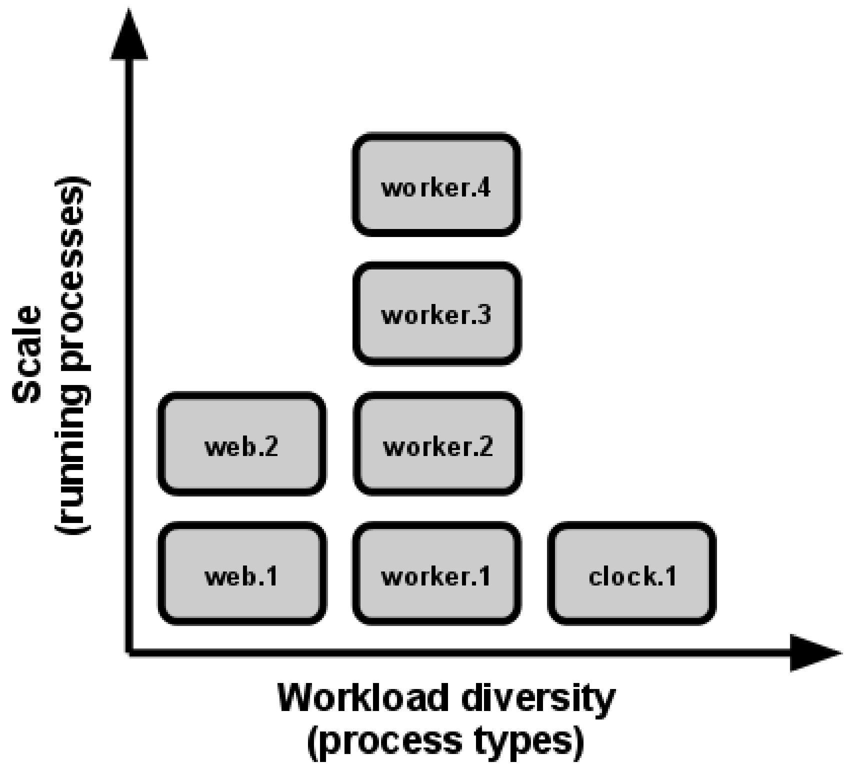

Figure 2.5: Process types / Concurrency

The ninth factor points that processes should be disposable, meaning they can be started or stopped at a moment's notice. Because of this, services can be started, stopped, or redeployed quickly. The shutdown should also happen gracefully after receiving a SIGTERM signal from the process manager, which means that no new requests are accepted, but existing ones are still handled. Robustness against sudden death is part of the factor, and it should be taken into account.[1]

The tenth factor is about having all environments as similar as possible. However, there can still be differences in the underlying data between environments. The twelve-factor app is designed for continuous deployment where gaps in time, personnel, and tools are as small as possible. To achieve this, developers should write code deployed hours or even minutes later, the same people who wrote the code should also deploy it, and no different backing services should be used between development and production.[1]

The second last factor, Logs, points that developers should not write code that concerns themselves with routing or storage of its log output stream. Each process writes into standard output, and one of the existing logging solutions should be used.[1] Logging is crucial for solving bugs, so it should also be taken into account when building applications.

The last factor explains that admin processes should be one-off processes executed alongside the application as a different process. These can be, for example, database migrations one-time scripts. These scripts should also be committed to the app's repository in order to avoid synchronization issues.[1]

3 Assessment of previous implementation

The architecture of the previously built application is traditional. Java application is running inside a Jetty server container, which communicates through Apache HTTP client. In Figure 3.1 the overall architecture is shown. The Bundled Fat JAR runs inside the Jetty Servlet Container, which runs inside the single virtual private server.

The frontend is built with an older version of Angular, which is no longer supported. The state management of the frontend is handled with the reactive programming library RxJS. It works in a similar way to the centralized data store.

Figure 3.2 demonstrates how BehaviorSubject and Observables work. The horizontal lines represent Observers that subscribe to BehaviorSubject. The subject then emits the most recent item it has observed and all subsequent observed items. Using this pattern avoids performance problems that come from passing state down to components linearly.

3.1 Twelve-factor analysis

Project handover was handled through secure file transfer, so there is not much to go on regarding the code repositories and deployments. Because of that, there is no way to know if the project had separate deployments for quality assurance or other

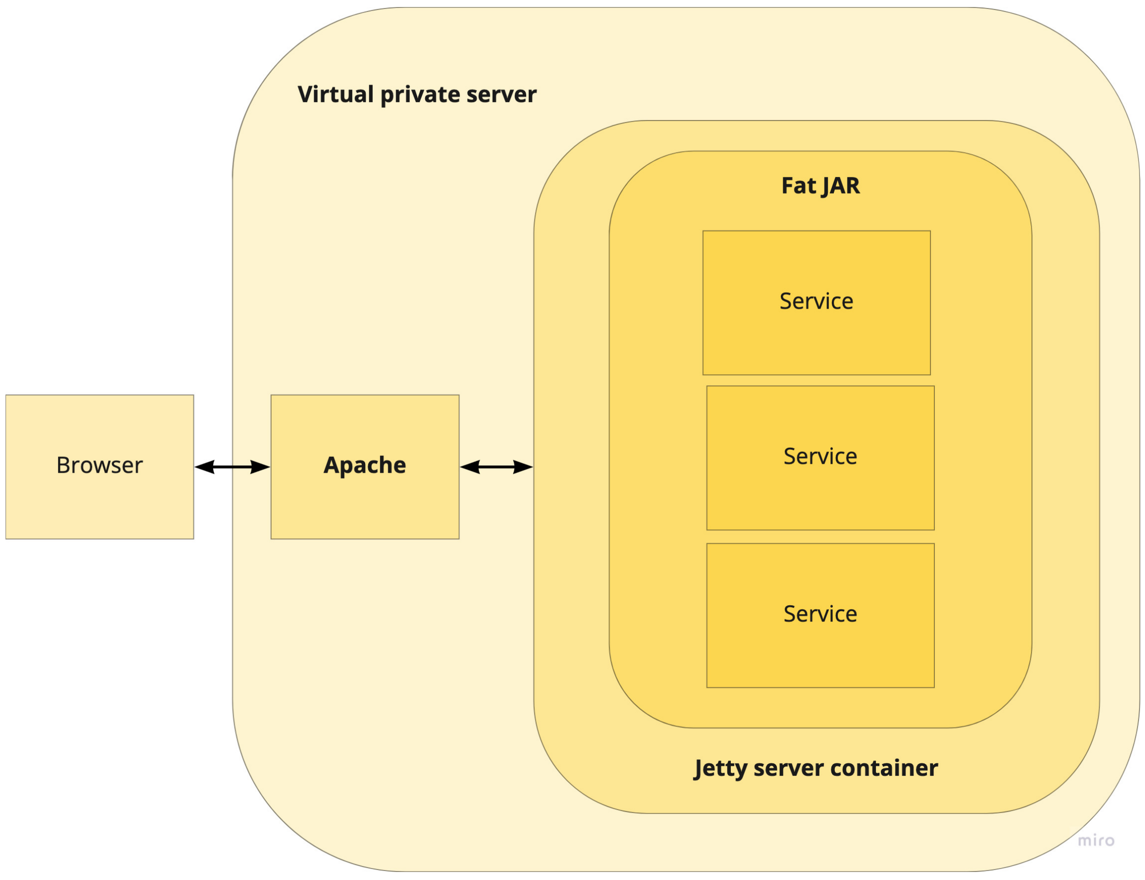

Figure 3.1: Previous implementation

stages. The only sure thing is that the product has one production deployment.

The project had explicit dependency declarations through manifests like build.graddle in backend and package.json in frontend when it comes to dependencies. This kind of approach has become more of a standard for building software. Actual bundling of dependencies is done with a Fat JAR or Uber JAR in the backend, which bundles the required together. This way, the app will not depend on any other dependencies coming from outside. Frontend does this in a similar way where Webpack bundles specified dependencies to one minified bundle.

Configs should always be separated from constants in the code. Some violations of this, however, can be found. Specifically, when it comes to storing API keys, they seem to be more like constants than environment variables. In this case, the

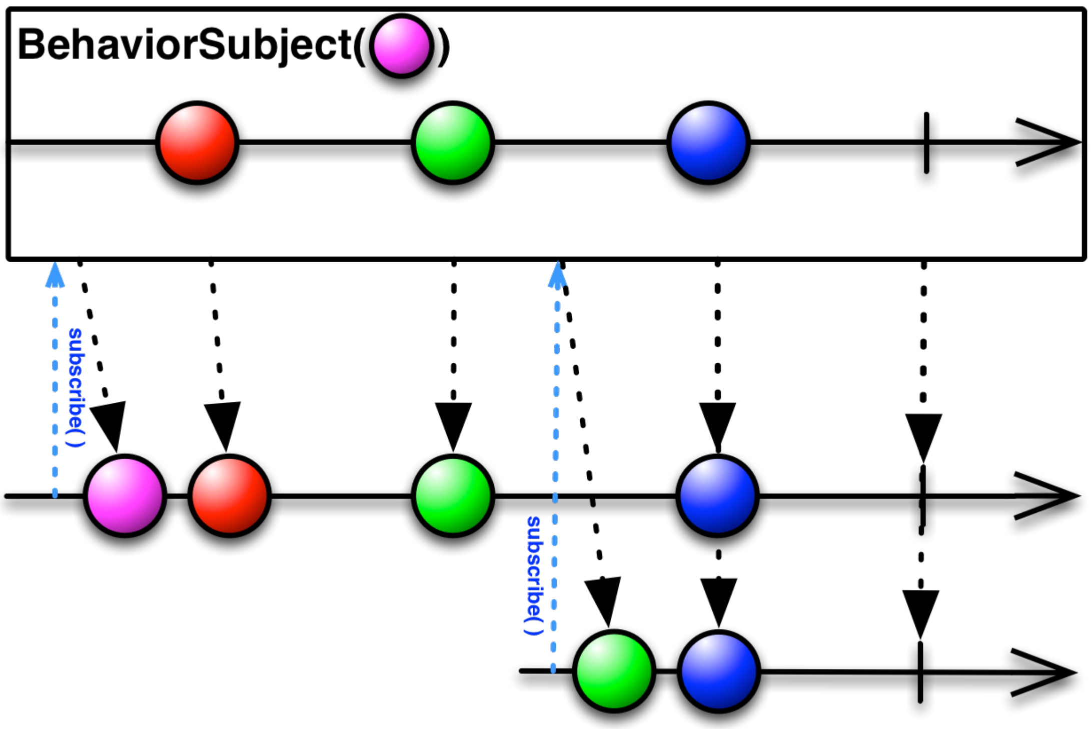

Figure 3.2: BehaviorSubject

app would not pass a litmus test to make the repository public. API keys to other systems are confidential, and they should not be readily available.

Backing services are correctly treated as resources without distinction between local services. The application uses several backing services, including external document management, mail, and local MySQL databases. Although the heavier integration to the document management system is more burdensome to replace, it is still just a swappable backing service since the rest of the app treats it as a similar resource like any other.

Build, release, and run factor is one of the more standardized approaches to making production-ready releases. The target application also conforms to this because it is bundled into a build and shipped together with config, making it a complete release. Every release also has its unique release id, which is composed of the current date and time. The actual pipeline could need some more work since the releases are stored in the server without much release management.

Running application process is stateless and shares nothing but a cache with the underlying system. This is a fairly standard approach where the actual process scaling is left to the server to handle. However, this kind of traditional server cannot scale up on-demand when scaling the application if the request amounts exceed server capacity.

The web app process exports HTTP as a service by listening to a port for incoming requests. This is also one of the standard ways to do RESTful APIs where the routing layer handles incoming traffic. In the case of this particular application, Jetty is used as a web server library.

In the backend, concurrency is managed by JVM overprocess, where a large chunk of the underlying system is reserved. Internally concurrency is managed by threads. This has its scaling problems with on-premises servers, as is mentioned before. However, scaling out via the process model is a good approach, and it does shine when the app has ways to scale out automatically.

Backend reacts to SIGTERM signals by invoking the stop server function, halting operations, and destroying the instance. This, however, is not a graceful shutdown since it does not process ongoing requests until the end and does not respond to incoming requests with error code 503 Service Unavailable. This way, the disposability factor is not fully taken into account.

Services themselves do not change based on development or production environment, so the Dev/Prod parity is preserved even in the original app. Dev/prod parity also encourages frequent deploys, but with the current setup without any existing pipelines, the actual production deploys can have a significant time gap between them.

Logging is implemented as an event stream writing into standard output with Log4j, a traditional logging service. This is the traditional way to handle logs, but the approach is not practical when reading them. This could be made much more convenient by sending the event stream to a cloud service.

Lastly, admin processes are handled as one-off tasks in the target application. Migration scripts have their folder where the administrator can pick and run them. These work the same in development and production environments. The only difference is that in production, developers need to use ssh to connect to the server.

3.2 Underlying infrastructure

There is no right solution while choosing infrastructure solutions, but there are always trade-offs. Most of the time, the choice is between managed cloud platform and an unmanaged private cloud. Managed means that the platform provider provides a service layer on top of the infrastructure. This includes change management, patch management, and health checking. Unmanaged has none of these services but instead has more freedom to manage the OS directly and choose the services they want to run.

The target system runs in a private cloud which is an unmanaged service. In this situation, the trade-off is the amount of maintenance the server needs. Private clouds need regular updates and much work when building automatic deployment pipelines. In the case of the target system, there are no additional services or pipelines. This leads to a situation where manual deployments are cumbersome and can lead to significant deploys infrequently.

3.3 Reasons for rewrite

The codebase of the previous implementation had not been updated for several years. Because of the infrequent maintenance, several of the main libraries used by the app were out of date. This was both a security concern and a problem for continuing development of new features.

Luckily, the size of the codebase was not large, but most parts of it needed updating. Some parts needed implementing entirely because of features that had not been finished. It was judged that updating the application would cost more than implementing a new one from the ground up.

4 New implementation

Driving factors when building the application were, first of all, robustness, being easy to maintain, being easily scalable, and security. Robustness came from the need to serve users efficiently. Scalability was a similar need that ensured that even more users could be served in the future. Being easy to maintain comes from the development side, where the focus is on long-term development and that the foundations of the app support future features.

Security is one of the driving points in decision-making. Recent news of security breaches increases the incentive to develop applications with security in mind. Several significant improvements were made, including blocking access to the database through SSH connections and having everything in the cloud.

4.1 Technologies

The architecture of the new implementation requires a wide range of technologies. For backend, used technologies are Kotlin, Ktor-framework, Gradle, Docker, Dotenv for secrets management, SLF4J for logging, JWT for authentication, Exposed as an ORM, and Gson for content negotiation. For frontend technologies include Typescript, React, CSS, SCSS, HTML, Dotenv, NPM, NVM, Axios for HTTP transactions, JSON, and React Context API to share data between components.

Finally, for infrastructure, Terraform was used to document and manage it. Amazon Web Services, also known as AWS, acts as a cloud provider that Terraform then manages. Added value from Terraform is that the infrastructure configurations are defined as code, and the changes can be tracked in version control which reduces human error and increases automation.

4.2 Application Architecture

The application is entirely built with AWS, and its infrastructure is managed with Terraform. AWS provides an advantage for building web applications because it has many ready-made services, reducing time invested into the application. Therefore to reduce costs, AWS was chosen.

In Figure 4.1 the entire architecture can be seen. It rests inside AWS and takes advantage of following services:

- Virtual Private Cloud or VPC
- Security Group or SG
- Relational Database Service or RDS
- Elastic Compute Cloud or EC2
- Internet Gateway
- Simple Storage Service or S3
- Cloufront
- Route 53

Starting from the top, VPC is a resource that is always needed. It is a logically isolated virtual network with its IP address range and subnets. The primary purpose of it is to allow developers to configure networking inside it. A security group is another related networking tool that acts as a virtual firewall and controls inbound and outbound traffic.

RDS houses the application database. It was chosen because it automatically handles hardware provisioning, database setup, patching, and backups. Out of the several available database engines, PostgreSQL was chosen because of its familiarity.

RDS is connected to EC2, where the backend logic is. It provides scalable compute capacity for the application so that capacity can be increased or decreased. EC2 contains an Ubuntu server, which the application runs on top of. Backend communicates to outside through Internet gateway, which allows communication between VPC and the internet.

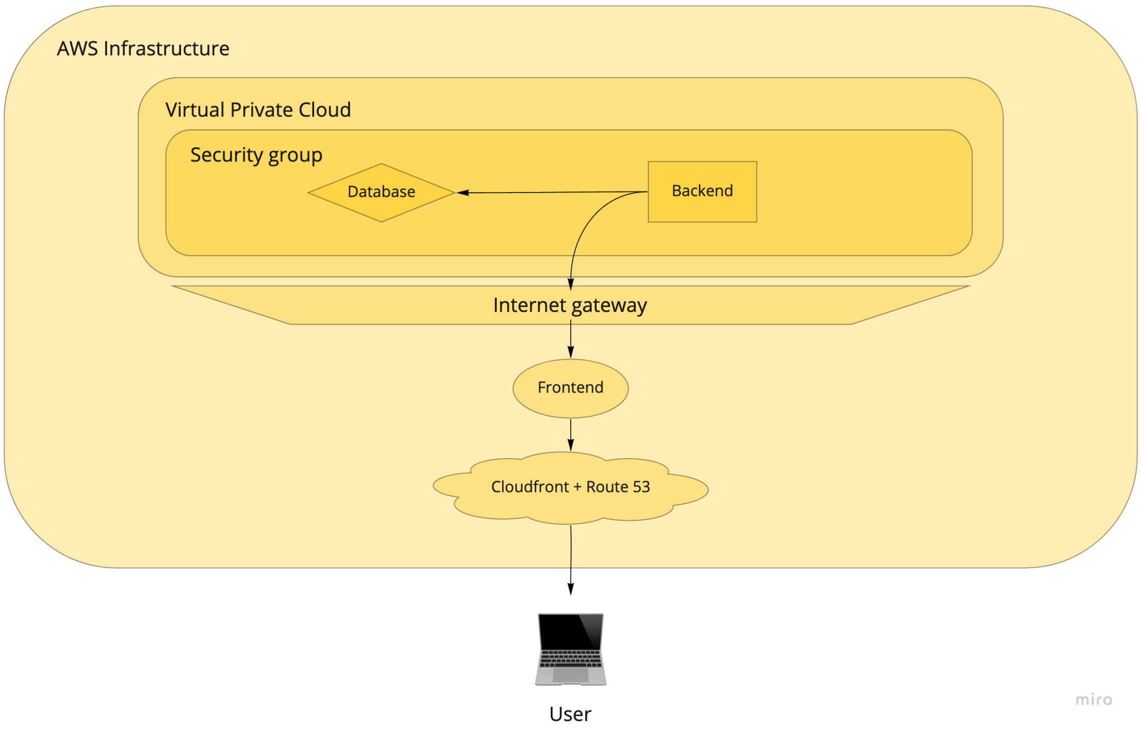

Figure 4.1: Architecture of the new implementation

Application backend is based on REST principles where the frontend requests a representation of the backend data.[25] The backend then handles frontend requests by issuing queries to the database. Like in Figure 4.2 the client requests data with an HTTP request, which triggers fetching, updating, or deleting of the particular data in the web service. Web service then performs queries to the database, after which the data is wrapped into JSON format and is sent to the client.

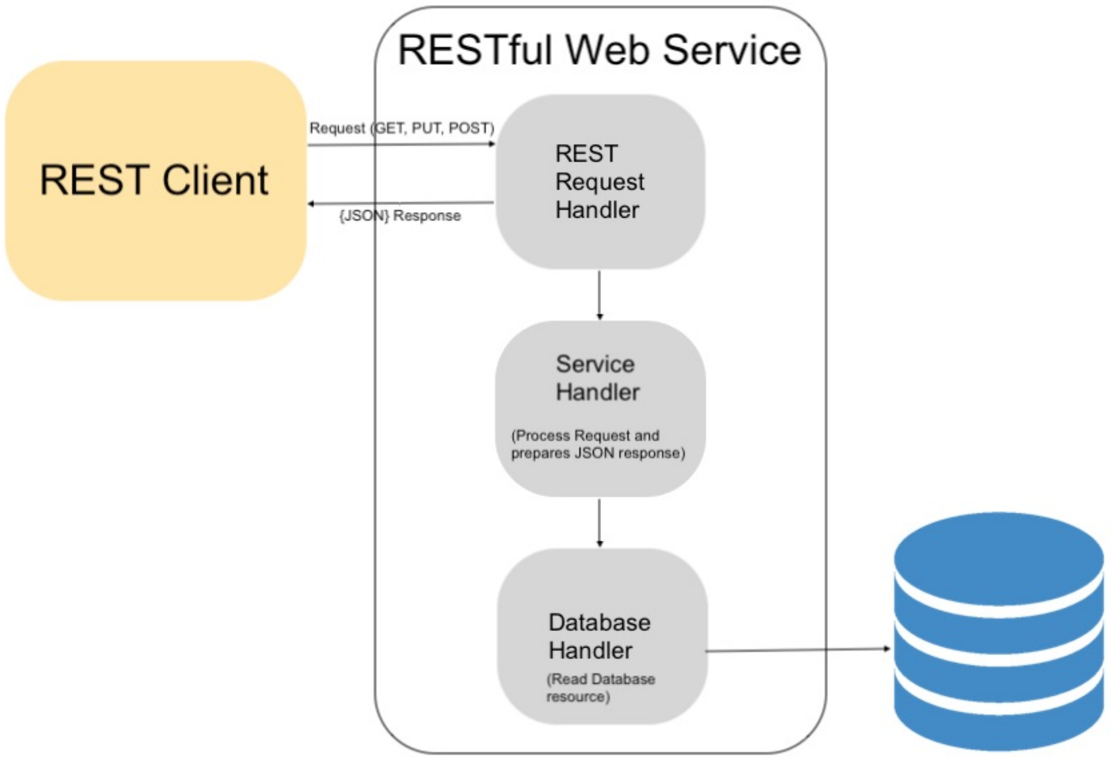

Figure 4.2: REST

The frontend is stored inside S3, which is an object storage service. From there, the application UI communicates with the backend asynchronously. To serve users efficiently, the application needs Cloudfront and Route 53 on top of S3. Cloudfront acts as a content delivery network, or CDN, which distributes the app globally through the AWS network backbone and provides an extra layer of security against DDoS attacks. Route 53 then translates domain names to infrastructure in our system, acting as a Domain Name System or DNS.

4.3 Frontend state management

Data is supplied to the frontend through the backend REST API. In this case, the data format is JSON, and the required data is retrieved in this format and requested with API calls. It is then supplied to a Context object that can be subscribed to, and then the value can be read.

Figure 4.3 describes state management between typical react application and react application that uses Context API. It shows that normally data is passed through the component tree at every level to a component that consumes the data. In Context, data consuming components can subscribe straight to the Context provider. Passing data or props at every level can be cumbersome because changing props triggers a re-render in the component.[26]

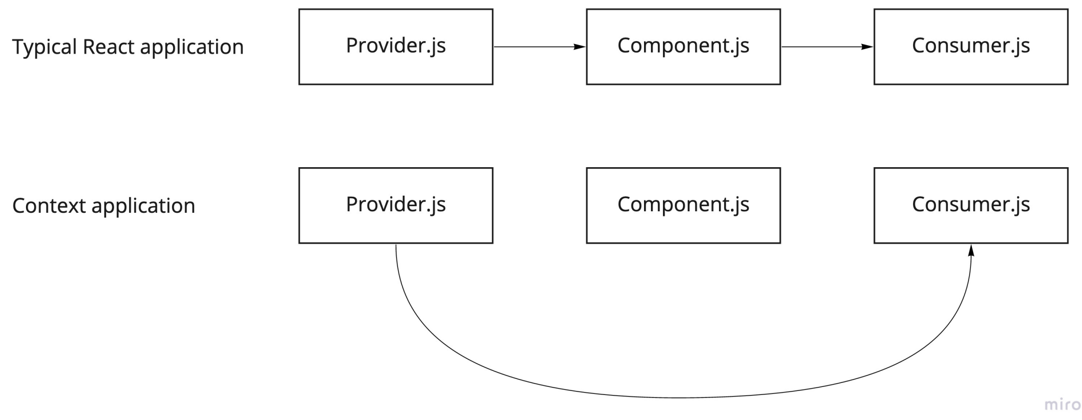

Figure 4.3: Typical react application vs. Context API

4.4 CI/CD pipeline

In order to deploy new versions of the app efficiently, Github Actions was used to create a CI/CD pipeline. In Figure 4.4 the pipeline is described. It triggers after a push to the main branch in GitHub, after which Gradle builds the application. It is then built into Docker image and shipped to Elastic Container Registry, or ECR. From there, EC2 pulls the image, restarts the server, and runs it.

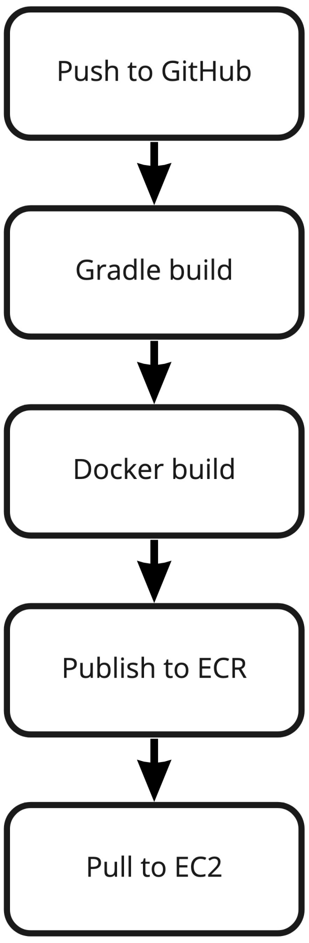

Figure 4.4: CI/CD pipeline for backend

Deploying the frontend is more straightforward, as shown in Figure 4.5. After pushing to the main branch in Github, the build command is issued to npm. The built application is then pushed to S3, which houses the frontend.

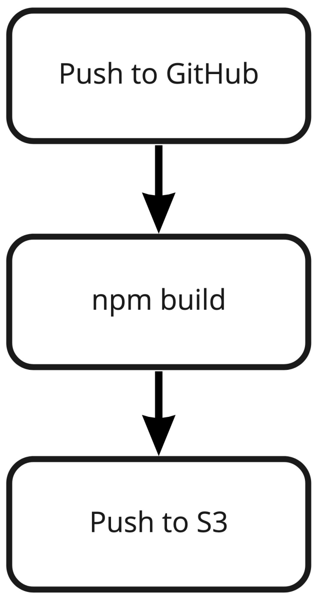

Figure 4.5: CI/CD pipeline for frontend

4.5 Twelve-factor analysis

The codebase is split into three repositories for clarity: Frontend, Backend, and Infrastructure. Usually, having huge mono repositories becomes hard to maintain and develop, so splitting them into parts based on domain makes sense. Github is used in this project as a version control service. Each developer can clone these repositories to their local machines to run them in their development environment.

Figure 4.6 shows how different repositories have their deployments. Infrastructure repository contains configurations in Terraform HCL language. Deployment infrastructure repository is different from backend and frontend since the actual infrastructure state is either in the local environment or deployed in AWS as a remote state.

Backend repository and Frontend repository have a more traditional deployment procedure where the application is run in the local environment and the production

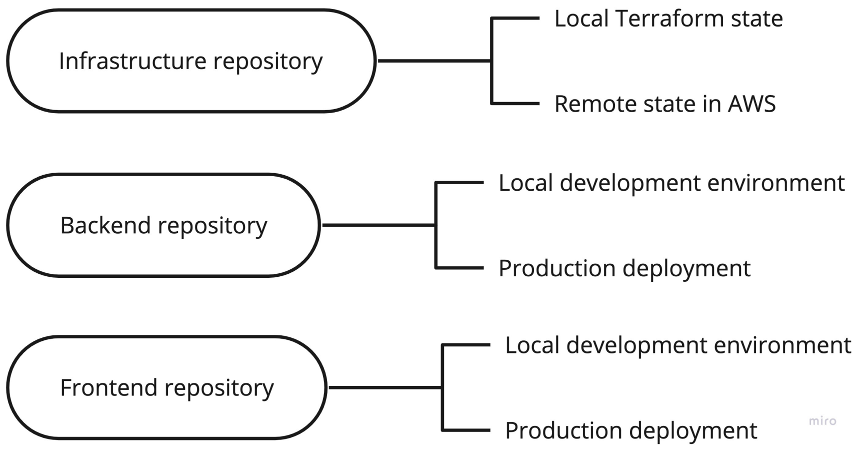

Figure 4.6: Repositories

environment. The frontend has its development server with a dedicated port from where it connects to the backend process through its port.

Like in the previous implementation, the replacement candidate also uses build.gradle and package.json manifest to define dependencies. The way bundlers and dependencies work has not changed much between these implementations, so the dependencies are the same. The replacement candidate uses the same approach as the previous implementation in the backend, where dependencies are bundled into one uber JAR.

Configs are handled with a module called Dotenv that loads environment variables from the .env file to process.env, keeping the Twelve-factor app requirement where the config is stored separately. Dotenv is used both in backend and frontend, but the environment variables are passed to the docker container in production.

In the new implementation, the backing services are indeed treated as third-party resources. Backend works as a REST API that provides data to the frontend. The backend also consumes data from a database, another service residing in the cloud.

The build stage follows a simple bundling process like in section 3. The backend is bundled into one JAR containing all of the dependencies, and the frontend is bundled into minified chunks. The backend is containerized with Docker, and environment variables are supplied to the container in the release process. Docker then runs the process in an isolated container.

The container process itself is completely isolated from the surrounding system because of the Docker container. The isolated container has its file system, networking, and process tree separate from the host, meaning that the factor six share-nothing policy is wholly fulfilled.

Port bindings work by instructing Ktor to listen to port 8080 like in listing 9. Then because the app is running inside a Docker container, we need to expose port 8080 by specifying EXPOSE 8080 in Dockerfile. When running the app in a cloud environment, we further need to expose that same port in the security group for requests.

By just having a web server listening to the port, we fulfill this factor. Because our implementation is using Docker, there are extra steps, and the additional security step is added when we restrict access from other destination ports.

Listing 9 Example of Ktor application.conf.

```
ktor {
    deployment {
        port = 8080
    }
}
```

Scaling out happens via the process model. Load balancer handles the scaling by ordering additional processes when the traffic goes up. There is, however, a cap on how many requests one Elastic Cloud can handle. When the server can no longer handle the traffic, it is possible to provision more servers. This way, AWS provides the possibility of horizontal scaling on top of vertical.

Disposing of these processes happens automatically since Ktor uses its implementation of a graceful shutdown. After receiving the generic signal for program termination, the backend process shuts down, and the frontend informs the user of an error. Because the frontend is separated and is, in this case, a static website that shows interactive and sends requests, the user can still get feedback even though the server is not running anymore.

The application follows the factor of having dev/prod parity between environments which means that the resources used do not differ between development and production. For example, the app uses the same Postgres database engine in both environments. Also, the data is similar, which helps reproduce errors found in production.

Another part of this factor is the time between deploys meant to be as short as possible. Understandably this should always be the goal: shipping small increments as frequently as possible. In this project, the goal was met only after the first release since the groundwork took some time. Otherwise, the requirements were met since having the same developer deploy the app was also fulfilled.

The final production-ready application handles logging by writing to standard output with logging library SLF4J[1]. Inside Docker, these logs can be accessed by typing docker logs which shows what is being written to the container's standard output. Future improvement could be to push these logs to AWS CloudWatch for better monitoring.

Admin processes like migrations are handled through one-time operations with SQL scripts. Scripts should be stored alongside the codebase, but there has not been any need to run these scripts because the app is recently built. It is indeed considered in the future that the migrations scripts are not in a different place.

[1]`http://www.slf4j.org/`

5 Results and evaluation

Twelve-Factor app methodology today is included in many implementations of frameworks and libraries. It has been nine years since the methodology was introduced. In programming, this is a long time because technology advances at a rapid pace. For example, after releasing Twelve-Factor in 2011, React, and Vue was released, which brought more competition and more ways to do frontend. Also EU General Data Protection Regulation was adopter in 2016, emphasizing security and careful handling of user data. However, the Twelve-Factor fundamentals have held up quite well.

5.1 Results

Twelve-Factor has become more or less the industry standard for making microservices and distributed applications. Especially while using container technologies like Docker, most of the factors can be fulfilled, such as Dev/prod parity, apps being executed as stateless processes, and scaling out via the process model. They are the natural extension of the Twelve-Factor app methodology and provide many of the factors as is. Table 5.1 shows that the original and the new implementation both cover most of the factors.

The previous implementation did not fulfill factors Config and Disposability. The new implementation fulfilled every factor. It should be noted that there are uncertainties about the Codebase factor because of the delivery method during handover.

Table 5.1: Implemented factors

Factor	1	2	3	4	5	6	7	8	9	10	11	12
Original	?	X		X	X	X	X	X		X	X	X
New	X	X	X	X	X	X	X	X	X	X	X	X

Factor three, Config, is often seen violated, just like in this case. Developers often mix environment configs, like for example, API keys and password specific to that environment, with regular constants that are not as critical when leaked, although it can be risky from a security standpoint because if the access to the codebase is compromised, then also the passwords and keys are leaked. Lastly, the difference between previous and new implementations regarding the ninth factor, Disposability, came to newer implementation using Ktor, which automatically handles the graceful shutdown.

When we look at the comparison in Table 5.2, some tooling differences can be found where the same goal is achieved but with a different tool. Older implementation has many solutions that implement factors by hand like separate build and run stages and Ktor implementing Disposability by processing ongoing requests, while newer implementation has tools that already implement the factor. Docker also boosts the efficiency of some factors like Processes, Concurrency, and Logs. CI/CD pipeline also keeps the Deployment gap small in the new implementation.

Because of the modern development tools that already implement the factors, it can be thought that parts of this methodology are no longer relevant for developers, although there is still value when looking at what parts have been abstracted away. Ideas like one codebase per service, using environment configs, backing services as attached resources, build, release, run, dev/prod parity, and admin processes being one-time tasks are relevant even today. However, others like Dependency declarations, Scaling out via process model, Port binding, Concurrency, Disposability, and

Table 5.2: Comparison of factors

No	Original	New
1	Not enough knowledge	GitHub repository
2	Clear declaration files	Clear declaration files
3	Parts of the config is inside the code	Config is in environment variables
4	Service is an attached resource	Service is an attached resource
5	Separate build and run stages	Separate build and run stages
6	Process is stateless but no autoscaling	Process is inside Docker container
7	Built-in Port binding with Jetty	Built-in Port binding with Ktor
8	Scales out via JVM	Scales out via Docker processes
9	Doesn't process ongoing requests	Does process ongoing requests
10	Good parity but has deployment gap	Good parity and the gap is small
11	Traditional server logfile	Docker stores the logs
12	One-off admin processes	One-off admin processes

Logs can be evident from the developer's point of view today. There are also shortcomings that this methodology does not address, like Authorization/Authentication, that should be standard practices today.

5.2 Proposal for a new model

In Kevin Hoffman's book Beyond the Twelve-Factor App[2] from 2016, three additions are introduced to the original Twelve-Factor App: API First, Telemetry, and Authentication/Authorization. API First and Telemetry are additions that we include in our model as-is, while Authentication/Authorization is grouped into a more general Security-factor to keep up with security demands. Also, a new factor has been added: Automation, which emphasizes the importance of freeing up developer time from mundane tasks.

New Ten-Factor App methodology would look like this:

1. Codebase
    - One codebase per service, tracked in revision control, many deploys.
2. Config
    - Store config in the environment.
3. Backing Services
    - Treat backing services as attached resources.
4. Build, Release, Run
    - Strictly separate build and run stages.
5. Dev/Prod Parity
    - Keep development, staging, and production as similar as possible.
6. Admin Processes
    - Run admin/management tasks as one-off processes.
7. API First
    - API should come first, and the rest of the feature is built around it.
8. Telemetry
    - Gather data about system performance and business domain on top of regular system logs.
9. Security

- Use TLS for securing data in transit, use API keys or JSON web tokens for Authorization/Authentication and follow the principle of least privilege while giving access.

10. Automation

- Automate mundane tasks like pipelines and code formatting to free up developer time.

Dependencies, Processes, Port binding, Concurrency, Disposability, and Logs, which Telemetry will replace, would be removed. Package managers most often handle dependencies, so there is little need to mention it as best practice. Port binding also comes included in frameworks like Ktor[1] and Spring boot[2] along with many others, which shows that it is no longer needed to include it as a best practice.

Processes, Concurrency, and Disposability are handled by containers which should be the default way to package applications. Container processes are stateless and immutable, and therefore they can be started concurrently by running several processes of the same container. Fast startup and graceful shutdown also come built-in with containers.[27] The final factor to be replaced is Logs which is expanded to general Telemetry of the application.

5.2.1 API First

API First is centered around the idea that services serve data and other services consume data. A contract documents services that provide data for consumers that the consumers can then depend on. Contracts can also be used to facilitate discussion about the needs of stakeholders since the concept of what data should be provided

[1]`https://ktor.io/docs/configurations.html`

[2]`https://docs.spring.io/spring-boot/docs/2.1.18.RELEASE/reference/html/howto-embedded-web-servers.html`

is more straightforward to discuss than the actual technical implementation.

Thinking in API-first approach is that API is the most important user of the application. Therefore when building features, the API should come first, and the rest of the feature is built around it. An added benefit is that some other team can start working on another feature centered around that API, providing faster Time to Market.

5.2.2 Telemetry

Telemetry includes factor 11 Logs, but on top of treating logs as event streams, it adds monitoring performance, domain-specific Telemetry, and system health monitoring to the factor. This new Telemetry factor is described in the book Beyond the Twelve-Factor App[2] as follows:

- Application performance monitoring (APM)
- Domain-specific telemetry
- Health and system logs

Application performance monitoring shows how the app is doing from an outside perspective, for example, how fast it responds to requests and how many requests the system can handle. Domain-specific Telemetry then focuses on collecting business-critical information like what products are selling well. Lastly, health and system logs show service-specific information about whether the service is operational and what is happening inside it. System logs especially should be redirected from the service for better accessibility.

Telemetry and especially Health and system logs monitoring is essential for cloud applications since errors happening in production need to be seen in a remote machine to be reproduced in the local environment. Also, service monitoring in the form of health checks can provide alerts when the service drops. In listings 10 and

11 it can be seen how basic health check can be implemented using Kotlin and Terraform. First, in Kotlin, specify a route, in this case, /health, and make sure it responds with status code 200. In this case, it just responds with the headers it received in the request. Then in Terraform, set load balancer in AWS to target previously defined route. Lastly, set desired thresholds and other configs.

Listing 10 Ktor route for health checks.

```
const val HEALTH_CHECK = "/health"

@Location(HEALTH_CHECK)
class HealthCheck

fun Route.health() {
    get<HealthCheck> {
        val headers = call.request.headers
        call.respond("Hello from server - received $headers")
    }
}
```

5.2.3 Security

Security can be a broad subject, but in this scope, the focus is on application development and what developers can do to improve security. A short checklist should look like this:

- Using TLS to secure data in transit (certificates)
- API keys or JSON web tokens for authorization and authentication
- Follow the principle of least privilege

Listing 11 Terraform configuration for health checks.

```
resource "aws_elb" "instance_lb" {

  health_check {

    healthy_threshold   = 2

    unhealthy_threshold = 10

    target              = "HTTP:8080/health"

    interval            = 30

    timeout             = 10

  }

}
```

The first item instructs to use TLS to secure data in transit and is from an article called Twelve-factor app development on Google Cloud[28]. Apps should use certificates to verify their authenticity and use HTTPS as default. The application described in Section 4 uses AWS Certificate Manager together with Amazon Route 53 to achieve this.

The next point is also from the same article: Use API keys or JSON web tokens for authorization and authentication. It means that access to the application should be controlled and planned from the start. For example, a route that provides business-critical data should be behind some authentication or authorization.

Lastly, the application should follow the principle of least privilege both in networking and authorization/authentication. Different layers of the app should only have access to the resources they need to function. In networking, traffic should be limited only to select IP ranges and ports. Especially on the networking side, this can reduce possible attack vectors.

5.2.4 Automation

The point of automation is to free up time for actual value-creating work. Today, there are many automation tools: Terraform for infrastructure provisioning, Github Actions for deployment pipelines, and ESlint and Prettier for static code analysis and code formatting. Automating mundane tasks should be a priority.

The automation factor is an extension to original factor ten, where the goal was to reduce personnel gap and time gap. Here, the idea is to reduce the personnel gap and follow DevOps best practices by encouraging team members not to outsource delivery but to do all the automation within the team. Teams themselves should build the infrastructure using IaC solutions and implement CI/CD pipelines for fast deployments, reducing the time gap.

6 Conclusion

In this thesis, an older application was replaced with a new implementation. The infrastructure of the older application was transferred to managed cloud provided by AWS instead of IaaS. The implementations and their architecture were described, and then both of the implementations were analyzed using the Twelve-Factor app methodology. Results were then evaluated, and we used Hevner's Information Systems Research Framework[3] to assess and evaluate our case study and formed new additions to our knowledge base in the form of Ten-Factor App best practice factors.

This thesis is limited to only one example case, but many of the technologies used can be applied to new implementations. Also, because the case study is used as the research method, results are not generalizable, and additional research is still needed to verify the results.

6.1 Discussion

Application architecture and infrastructure are becoming more and more intertwined because of Infrastructure as Code services. Almost the same people are building products from frontend to hardware provisioning, which helps with knowledge sharing in software development teams. An added advantage of IaC is reduced costs and speed of deploying resources.

Knowledge needed for the average software developer is also increasing because of IaC. Software projects need experts that roughly understand all of the parts needed to deploy the full product. Deep knowledge of all aspects might not be necessary, but broad understanding helps with problem-solving. The Ten-Factor app methodology factors described in this thesis give a good start on what should be considered.

Hasura's 3factor app[1] proposes a different kind of general solutions for a software product. In a similar way that Twelve-Factor collects best practices into twelve factors, 3factor combines three points into what they envision software product architecture should look like. It abstracts away details that twelve-factor describes, so they are not entirely mutually exclusive models.

3factor relies on three points: real-time GraphQL, Reliable eventing, and Async serverless. Realtime GraphQL means that any data should be fetched with a single call without blocking or polling on synchronous requests. Reliable eventing then points out that the event system should invoke business logic and that events should be created when the state changes while also having at least once guaranteed delivery. Lastly, Async serverless instructs to use serverless backends for business logic where the code can handle duplicate events and events that do not come in order.

3factor app brings something new entirely to the table, and while it is not a direct successor, it is still an interesting next step. All of the previously mentioned practices still apply even with 3factor. It should be noted that while technical tools evolve, collaboration, communication, and willingness to learn remain as cornerstones of software development.

6.2 Future research

The Twelve-Factor app has inspired different variations after its writing. One of them is the 3factor app, an architecture pattern for modern full-stack apps, made by Hasura[2]. The pattern proposes that all business logic is invoked via events, and

[1]`https://3factor.app/`

[2]`https://hasura.io/`

state management should be removed from the API layer. It would be interesting to see comparisons between old apps built traditionally and apps built with 3factor design.

References

[1] A. Wiggins. (). “The twelve-factor app”, [Online]. Available: https://12factor.net/. (accessed: 02.06.2021).

[2] K. Hoffman, *Beyond the Twelve-factor App: Exploring the DNA of Highly Scalable, Resilient Cloud Applications*. O’Reilly Media, 2016.

[3] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research”, *MIS quarterly*, pp. 75–105, 2004.

[4] A. Håkansson, “Portal of research methods and methodologies for research projects and degree projects”, in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*, CSREA Press USA, 2013, pp. 67–73.

[5] MDN contributors. (). “A re-introduction to javascript”, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript. (accessed: 17.01.2021).

[6] Microsoft. (). “What is typescript?”, [Online]. Available: https://www.typescriptlang.org/. (accessed: 17.01.2021).

[7] Facebook Open Source. (). “Components and props”, [Online]. Available: https://reactjs.org/docs/components-and-props.html. (accessed: 26.06.2021).

[8] ——, (). “Introducing jsx”,
[Online]. Available: https://reactjs.org/docs/introducing-jsx.html.
(accessed: 26.06.2021).

[9] ——, (). “File structure”, [Online]. Available:
https://reactjs.org/docs/faq-structure.html. (accessed: 26.06.2021).

[10] Google. (). “What is angular?”, [Online]. Available:
https://angular.io/guide/what-is-angular#what-is-angular.
(accessed: 26.06.2021).

[11] Jetbrains. (). “Kotlin”,
[Online]. Available: https://kotlinlang.org/. (accessed: 26.06.2021).

[12] ——, (). “Null safety”, [Online]. Available:
https://kotlinlang.org/docs/null-safety.html. (accessed: 26.06.2021).

[13] ——, (). “Getters and setters”, [Online]. Available:
https://kotlinlang.org/docs/properties.html#getters-and-setters.
(accessed: 26.06.2021).

[14] K. Kolyshkin, “Virtualization in linux”,
White paper, OpenVZ, vol. 3, no. 39, p. 8, 2006.

[15] Docker Inc. (). “Docker overview”,
[Online]. Available: https://docs.docker.com/get-started/overview/.
(accessed: 26.06.2021).

[16] Git. (2020). “Getting started - about version control”,
[Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. (accessed: 17.01.2021).

[17] Amazon Web Services, Inc. (). “What is devops?”,
[Online]. Available: https://aws.amazon.com/devops/what-is-devops/.
(accessed: 19.06.2021).

[18] Redhat. (). "What are cloud services?", [Online]. Available: https://www.redhat.com/en/topics/cloud-computing/what-are-cloud-services. (accessed: 20.01.2021).

[19] S. Bhardwaj, L. Jain, and S. Jain, "Cloud computing: A study of infrastructure as a service (iaas)", *International Journal of engineering and information Technology*, vol. 2, no. 1, pp. 60–63, 2010.

[20] Amazon Web Services. (). "Ec2", [Online]. Available: https://aws.amazon.com/ec2/. (accessed: 26.06.2021).

[21] HashiCorp. (). "Introduction to terraform", [Online]. Available: https://www.terraform.io/intro/index.html. (accessed: 26.06.2021).

[22] C. Richardson. (). "Pattern: Microservice architecture", [Online]. Available: https://microservices.io/patterns/microservices.html. (accessed: 04.06.2021).

[23] R. C. Martin, "The single responsibility principle", *The principles, patterns, and practices of Agile Software Development*, pp. 149–154, 2002.

[24] B. Horowitz. (). "Mra, part 5: Adapting the twelve-factor app for microservices", [Online]. Available: https://www.nginx.com/blog/microservices-reference-architecture-nginx-twelve-factor-app/. (accessed: 03.06.2021).

[25] R. T. Fielding, *Architectural styles and the design of network-based software architectures.* University of California, Irvine, 2000.

[26] Facebook Open Source. (). "Context", [Online]. Available: https://reactjs.org/docs/context.html. (accessed: 16.06.2021).

[27] Docker Inc. (). "Docker-compose up",
[Online]. Available: `https://docs.docker.com/compose/reference/up/`. (accessed: 27.06.2021).

[28] Cloud Architecture Center. (). "Twelve-factor app development on google cloud",
[Online]. Available: `https://cloud.google.com/architecture/twelve-factor-app-development-on-gcp`. (accessed: 06.05.2021).