Final Year Project

Performance Analysis of Design Patterns in Microservice Architecture

Rajit Banerjee

Student ID: 18202817

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Professor John Murphy

UCD School of Computer Science

University College Dublin

11th April 2022

Contents

Abstract

TODO Provide a short description here (about 150-200 words) of your project. Do not go into fine detail, but offer a strategic overview of the project's aims.

Chapter 1: Project Specification

1.1 Problem Statement

Microservice architecture is a style of designing software systems to be highly maintainable, scalable, loosely-coupled and independently deployable. Moreover, each service is built to be self-contained and implement a single business capability. Design patterns in software engineering refer to any general, repeatable or reusable solution to recurring problems faced during the software design process. The aim of this project is to study the performance engineering practices associated with a number of *microservice design patterns*, considering both qualitative and quantitative metrics to evaluate their benefits and shortcomings depending on the business requirement and use case. A non-exhaustive list of design patterns that could be explored is as follows:

- API Gateway
- Asynchronous Messaging
- Chain of Responsibility
- Database or Shared Data
- Circuit Breaker
- Externalise Configuration
- Aggregator
- Branch

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

1.2 Background

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

1.3 Related Work

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

1.4 Datasets

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

1.5 Resources Required

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

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

Chapter 2: Introduction

TODO Introduce your vision of the project here. Describe the domain of the project, and the intended application. A well-written report will answer three key questions: What am I doing in this project? Why is it worth doing? How do I plan to go about it? In this introductory section, offer a concise answer to the What, and follow-up with a compelling account of the Why. Leave the How to a subsequent section. Do not try to do too much in any single section of the report. By providing details in a logical order, you will show that you have a plan for the report and the project.

Chapter 3: **Related Work and Ideas**

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

3.1 Microservices

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

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

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

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

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

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

3.2 Software Design Patterns

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

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

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

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

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

3.3 Common Design Patterns in Microservice Architecture

Several attempts have been made to apply the concepts of software design patterns to microservices and categorise commonly seen patterns. In Fig. 3.1, C. Richardson provides a number of pattern groups, including *Decomposition, Data management, Transactional messaging, Testing, Deployment, Cross-cutting concerns, Communication style, External API, Service discovery, Reliability, Security, Observability* and *UI* [21].

Similarly, M. Udantha describes 5 different classes of design patterns applicable to microservices, namely *Decomposition, Integration, Database, Observability* and *Cross-cutting concerns* (see Fig. 3.2).

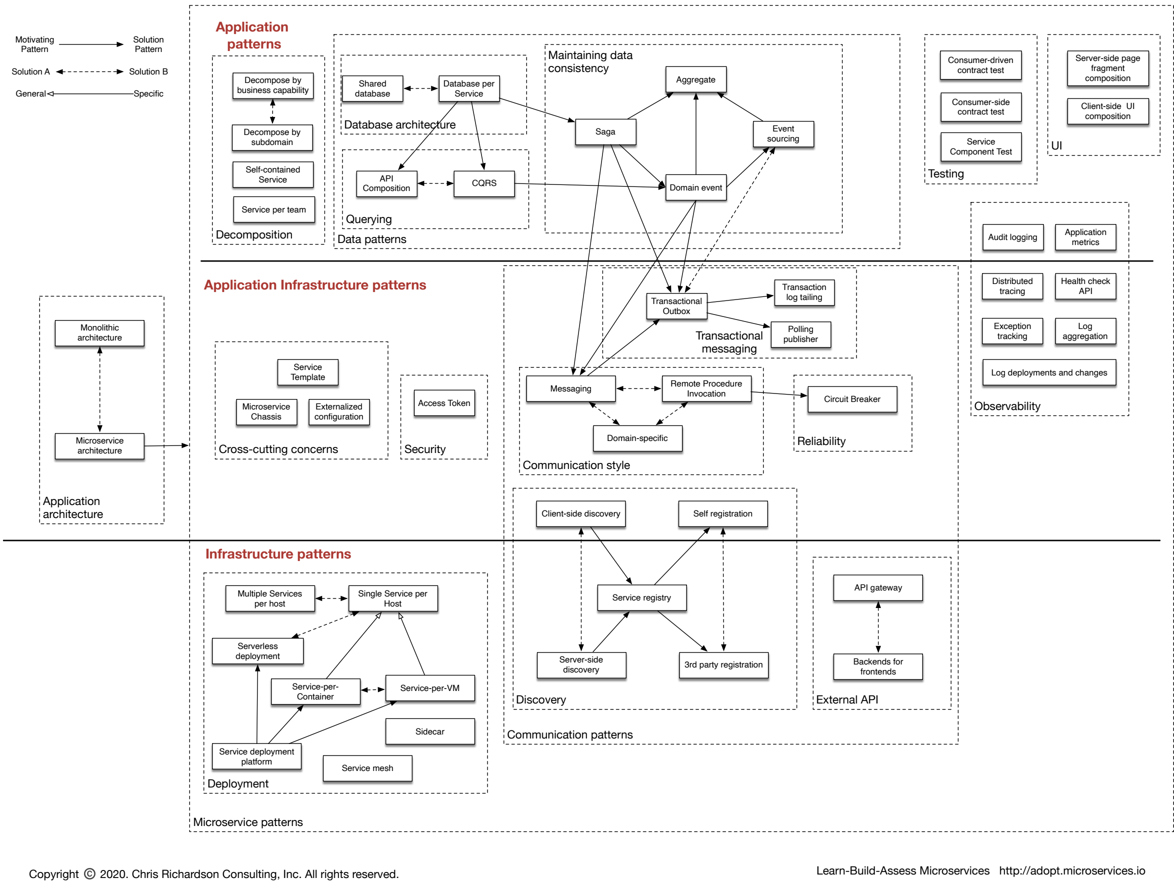

Figure 3.1: Groups of microservice design patterns [21].

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

- *Decomposition* patterns lie at the heart of microservice design, and illustrate how an application can be broken down by business capability, subdomain, transactions, developer teams, etc. They also include refactoring patterns that guide the transition from monoliths to microservices.

- *Data management* patterns guide the design of database architecture (e.g. whether multiple services will share a database or each service will get a private database). They also lay out methods for maintaining data consistency, dealing with data updates and implementing queries.

- *Integration* patterns include API gateways, chain of responsibility, other communication mechanisms (e.g. asynchronous messaging, domain-specific protocols), as well as user interface (UI) patterns.

- *Cross-cutting concern* patterns describe ways of dealing with concerns that cannot be made completely independent, and result in a certain level of tangling (dependencies) and scattering (code duplication). Examples include externalising configuration, handling service discovery (client-side such as Netflix Eureka [5]; server-side like AWS ELB [6]), using circuit

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

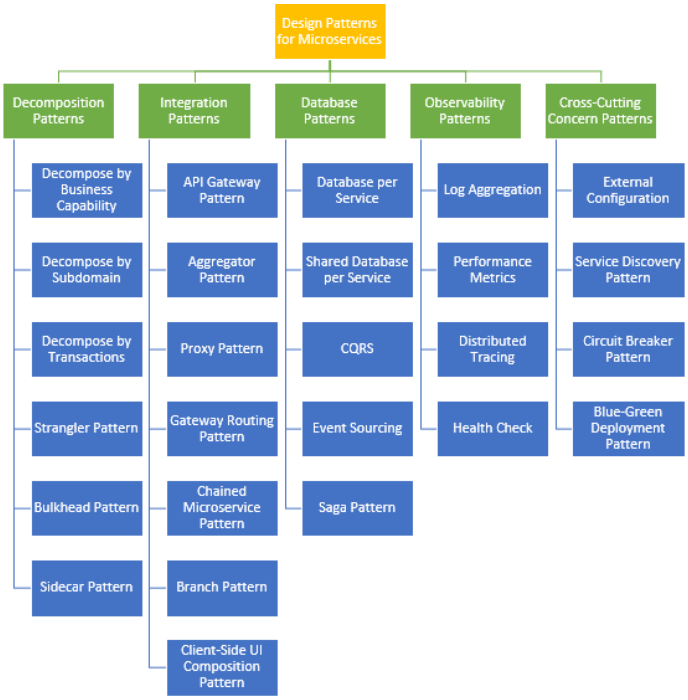

Figure 3.2: Udantha's 5 classes of microservice design patterns [8].

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

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

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

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

3.4 Issues and Challenges with Microservices

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

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

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

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

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

3.5 Performance Engineering

Having discussed the background research and ideas related to microservice-based systems and design patterns, it is now appropriate to narrow down the focus to performance engineering for distributed systems, especially for microservices and their design patterns. Performance may be defined as the extent to which a system meets its timeliness objectives, and the efficiency with which this is achieved. Software performance engineering (SPE) is a vital part of the software design lifecycle, including, but not limited to performance simulation and modelling (e.g. using UML [7] diagrams), benchmarking, monitoring, and performance testing.

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

3.5.1 Distributed Systems

Innovation in performance engineering has often been driven by breakthroughs in industry, out of necessity. In fact, Netflix is well known for pioneering the concept of *chaos engineering* as early as 2011, when migrating to the cloud (AWS). To address the inadequacy of available resilience testing, the company invented a suite of tools (known as the "Simian Army" [31]), the most significant of which is Chaos Monkey [8]. Chaos engineering tests the performance of a large-scale system when subjected to experimental failure scenarios, especially in production environments.

Similarly, there exists the principle of *ownership* at Amazon, where it is the development team's responsibility to handle the entire software lifecycle including operations, performance testing, and monitoring ("you build it, you run it" [32]). Some contributions from academia include the introduction of a policy-based adaptive framework to automate the usage of diagnosis tools in the performance testing of clustered systems by Portillo-Dominguez et al. [33], [34]. The authors refer to the increased complexity of performance testing in distributed environments and provide a framework (PHOEBE) to increase time savings for testers.

3.5.2 Evaluation of Microservice-Based Systems

The 2015 study by Amaral et al. [35] stands out as the first work of its kind, as claimed by the authors: to analyse the performance of microservice architectures using containers (specifically Docker [9]). The paper provides a comparison of CPU performance and network running benchmarking for two models of designing microservices: master-slave and nested-container.

In 2017, Heinrich et al. explored the performance engineering challenges specific to microservices, namely testing, monitoring and modelling, with a focus on the need for efficient performance regression testing techniques, and the ability to monitor performance despite regular updates to software [36]. The paper identifies open issues and outlines possible research directions to tackle the lack of adequate performance engineering for microservices. In the same year, Gribaudo et al. took a simulation based approach to study the behaviour of microservice-based software architectures, and used a randomly generated (and realistic) overall workload to evaluate infrastructure setup, performance (defined indexes such as response time) and availability. The authors claim that their work is the first parametric simulation approach for performance modelling of microservices [37].

In more recent studies (2020), Eismann et al. discuss both the benefits and challenges introduced by microservice architectures from the perspective of performance testing [38]. Containerisation, granularity, access to metrics and DevOps integration are mentioned as characteristics that aid performance testing, whereas some pitfalls such as lack of environment stability, experiment reproducibility and detecting small changes for performance regression testing are also present. Gias et al. look at tailored performance engineering techniques for cloud-native microservices, which are increasingly becoming the industry norm due to the various benefits of cloud computing [39]. The authors use the RADON [10] project to address performance modelling challenges in microservices and FaaS (Function as a Service, such as AWS Lambda [11]), then go on to address aspects such as deployment optimisation, continuous testing and runtime management.

Finally, the primary inspiration for this project is the 2019 paper by A. Akbulut and H.G. Perros who take a closer look at 3 distinct microservice design patterns: API Gateway, Chain of Responsibility and Asynchronous Messaging, and compare performance analysis results (in terms of query response time, efficient hardware usage, hosting costs and packet loss rate) with different hardware

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

[9] https://www.docker.com

[10] Rational Decomposition and Orchestration for Serverless Computing

[11] https://aws.amazon.com/lambda/

configurations (e.g. virtual CPU and RAM) [11]. Some results from the first case study on the API Gateway pattern are shown in Fig. 3.3.

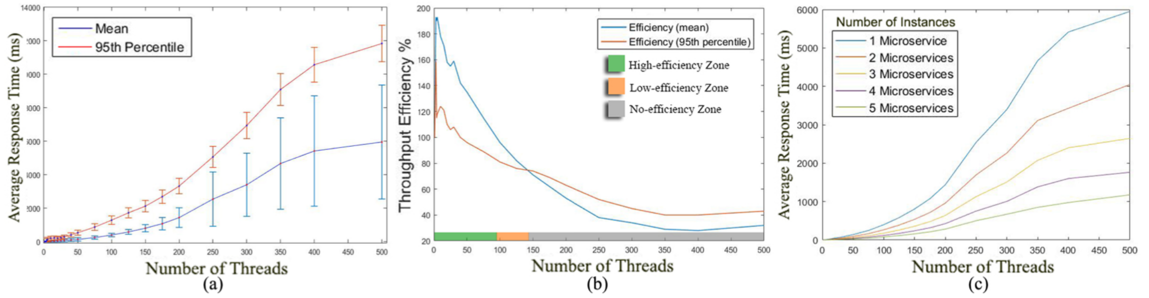

Fig. 1. Performance Results of Case Study I a)Mean and 95th Percentile of the Response Time vs Number of Threads, b)Efficiency Curves Using the Mean and the 95th Percentile of the Response Time, and c)Average Service Times for Different Number of Instances

Figure 3.3: Results from Akbulut and Perros' case study on the API Gateway design pattern [11], showing the variation in performance metrics with increasing number of threads used to generate asynchronous requests.

At the time of writing, Akbulut and Perros' work was found (to the best of our knowledge) to be the only paper to consider the performance analysis of design patterns for microservices, and this project aims to possibly replicate some of their results, then extend the study to other useful design patterns beyond the 3 that the authors have evaluated. This project will also take a similar approach by designing microservice-based applications using Docker containers, the performance engineering aspects of which will be studied.

3.6 Summary

The background research conducted reveals that in under a decade, microservices have gone from a nascent architectural style to becoming the de facto choice for building large-scale, flexible, loosely-coupled and maintainable software in the industry. Design patterns are templates for best practices that were initially applied to software design in general, but have been adapted for microservice architecture as well. They often guide the process of refactoring a system from a monolith to fine-grained services, considering aspects such as decomposition strategy, communication style, distributed data management, observability, and much more. Still, there are a number of hurdles to cross and anti-patterns to avoid when adopting microservices, and a proper needs assessment must be conducted before making the choice to disregard a well-designed monolith. Finally, performance engineering is a key contributor towards the success of any software, and is especially challenging and important for distributed systems such as microservice-based systems. The limitations of existing works include the lack of adequate performance analysis for various microservice design patterns, which has only been tackled by Akbulut and Perros [11] so far. This provides the motivation for proceeding with this project to extend the related works.

Chapter 4: Outline of Approach

4.1 System Overview

In order to demonstrate various performance engineering practices in the context of microservice design patterns, we look at a simple movie ticket reservation system, consisting of 3 independent cinemas (Cineworld [1], Dundrum [2] and UCD [3]), as well as an intermediary (or broker) between clients and the cinema services. Each cinema has its own catalogue of movies (with an ID, name, and available showtimes). The client's primary objective would be to request the intermediary to fetch a list of movies from every active cinema, then display aggregated results. Next, the client selects a movie and showtime (along with other booking details) to make a reservation at a specific cinema. It is also possible to list the reservations made at a given cinema (useful for staff and administrators). The intermediary serves as a router for client requests to the cinema services, and in some cases, an aggregator of results.

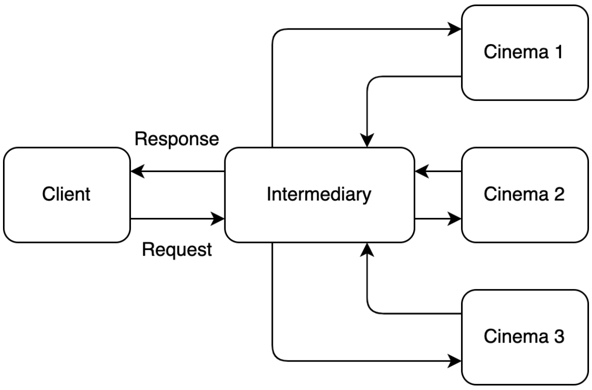

Figure 4.1: Prototype of movie ticket reservation system using microservices.

4.2 Case Studies

In this project, two separate web applications have been designed to model the same system described above, each employing a variety of common design patterns. The applications follow microservice architecture, with the intermediary and cinema services all being self-contained, loosely-coupled and independently deployable. Taking each application to be a *case study*, the next chapter provides descriptions of the system design, implementation specifics, design pattern choices, and the associated performance implications. In a larger scale production system, it is likely that each independent cinema would in turn comprise a number of microservices to serve more specific purposes, but to avoid over-complicating the architecture in this project, cinemas are designed to be single entities.

[1] https://www.cineworld.ie/

[2] https://www.movies-at.ie/movies/

[3] https://www.ucd.ie/studentcentre/cinema/

4.2.1 Communication Styles

When migrating from monolithic applications to microservices, decisions regarding communication mechanisms are vital, since a number of internal services must efficiently interact with each other as well as the clients, in order to perform well in a distributed environment. Managing data from responses, and aggregating results from various services also becomes a part of the equation. The two case studies in this project illustrate the two most common communication style patterns, namely HTTP [4] request/response using APIs (typically RESTful [5]) in case study 1, and asynchronous messaging following a protocol like AMQP [6] in case study 2. Broadly speaking, the first style uses a synchronous protocol (HTTP), where the client awaits a response from services after sending a request. Then, as the name suggests, the second style uses an asynchronous protocol (AMQP) that uses message queues, and doesn't wait for responses. These have been separated from demonstration, but in practice, microservice-based applications tend to employ a combination of synchronous and asynchronous communication styles (including deferred synchronous and asynchronous callbacks) depending on the use case. Microservices often face the issue of "chattiness", due to excessive inter-service communication leading to performance degradation, which is a point to keep in mind when developing microservices. The aim is to enforce the autonomy of microservices during integration, and avoid anti-patterns such as a distributed monolith, where services end up being tightly coupled.

4.2.2 Choice of Tools

One of the main benefits of microservices is the freedom of choice regarding the programming language used for development, since any suitable language may be used as long the service exposes an API [7] adhering to known standards (typically HTTP and REST). The API is then used for inter-service communication. For the case studies in this project, Java and Spring [8] are chosen over other languages and frameworks primarily due to the dominance of Java in enterprise-level software, as well as the feature richness of Spring. Moreover, as shown in a 2017 study by Pereira et al. [40], Java ranks much higher in terms of time, memory and energy efficiency; which need to be considered when talking about system performance; compared to other popular languages such as Python or TypeScript/JavaScript. The Spring framework in Java offers a plethora of integrations and abstractions to implement microservice and cloud-related design patterns, through projects such as Spring Boot, Spring Data, Spring Cloud and Spring AMQP, to name a few that are used in the case study web applications here. Apache Maven [9] is the tool used for dependency management for Java.

Each web application exposes APIs for its microservices, whose endpoints can be invoked with HTTP requests like GET and POST. In commercial systems, a website (UI client) would invoke the backend API to perform actions, but for our case studies, it is sufficient to use a tool such as Postman [10] or VS Code's REST Client [11] to demonstrate the API functionalities locally.

[4] https://developer.mozilla.org/en-US/docs/Web/HTTP
[5] https://en.wikipedia.org/wiki/Representational_state_transfer
[6] https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
[7] https://www.mulesoft.com/resources/api/what-is-an-api
[8] https://spring.io
[9] https://maven.apache.org
[10] https://www.postman.com
[11] https://marketplace.visualstudio.com/items?itemName=humao.rest-client

4.3 Deployment

In line with industry standards, Docker [12] is the infrastructure/containerisation tool used to deploy the fleet of microservices. According to Docker, Inc., a *container image* is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Using Docker provides a number of advantages such as ease of standardisation, compatibility, maintainability, CI/CD, resource and application isolation, and much more. Container deployment and application testing is performed on an Ubuntu 18.04 LTS compute server (*dunnion*, maintained by the UCD School of Computer Science [13]), with a 10 core Intel(R) Xeon(R) Silver 4114 CPU (base frequency: 2.20 GHz), and 125GB system memory.

4.4 Evaluation

[12] https://www.docker.com

[13] https://www.ucd.ie/cs/

Chapter 5: Case Studies

5.1 Case Study 1

The pilot case study implements the movie ticket reservation system prototype described in the outline earlier, primarily using microservices communicating with REST APIs. Representational State Transfer implies that a client requesting a resource from any service, cues the server to transfer back the current state of the resource in a standardised representation. The system design and implementation details are elaborated below.

5.1.1 Design and Implementation

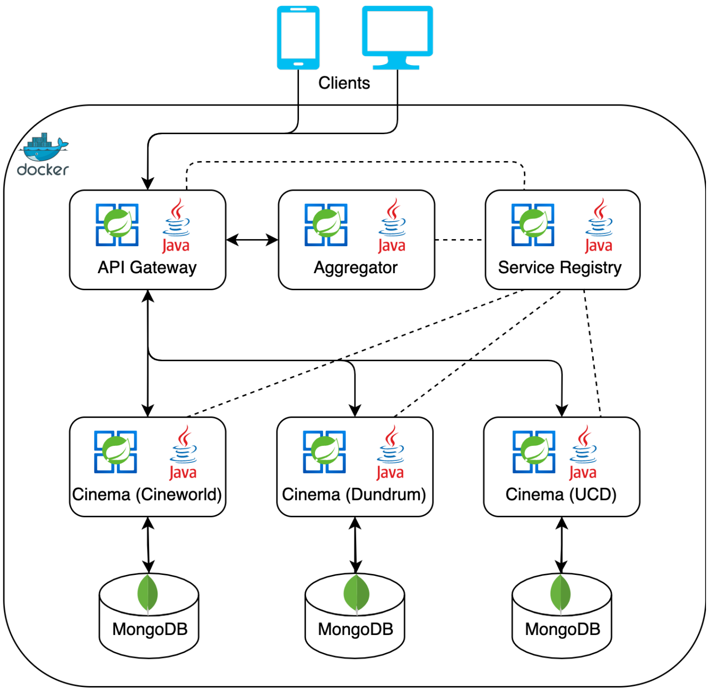

Figure 5.1: System design for the first case study.

The architecture diagram above (Fig. 5.1) depicts the organisation of the microservices involved: an API gateway service to act as an interface between clients and other services, and aggregator service to gather results, a central service registry, and three independent cinema services - Cineworld, Dundrum and UCD. The following microservice design patterns have been employed while developing this web application:

API Gateway

An *API gateway* is one of the most fundamental and commonly used design patterns in microservice architecture. The `api-gateway-service` microservice offers an entry point for all clients, so that multiple services (e.g. cinemas) can be exposed, and gateway can route requests to other services as appropriate. In such situations, an API gateway relieves the clients from setting up and managing a separate endpoint for every service. It also aids the inter-service communication between microservices when required, for instance, the `aggregator-service` contacts the cinema services via the gateway again.

Without a gateway, any change (e.g. refactoring) in the service APIs would necessitate corresponding changes in the client code as well, which creates several development and management troubles. However, when a gateway is placed between clients and services, further consolidation or decomposition of a service would only require a simple routing change, instead of updating the client.

An added benefit of this pattern would be the ease of deployment of new versions of microservices, which can be done in parallel with existing versions, since API gateway routing would provide control over endpoints presented to clients, and flexibility over feature release strategies.

A minor variation of the API gateway pattern is called *backends for frontends (BFF)*, where a separate gateway handles request routing for different kinds of clients, such as web applications, mobile applications and third part applications (e.g. IoT devices). API endpoint management in appropriate gateway services is a more focussed approach to handling routing logic for different kinds of user interfaces, with their own set of pre-requisites.

Despite its popularity and usefulness, the API gateway pattern brings with it a few issues that should be taken into consideration. For instance, it could become a bottleneck or single point of failure, if resiliency and fault tolerance measures are not put in place. Performance testing methods such as load testing should be explored to prevent failure scenarios.

In this case study, the `api-gateway-service` uses the Spring Cloud [1] project to introduce a gateway [2] in the application. The gateway is configured using the module's `application.yml` resource file, which Spring Boot [3] can access to set up all the routing logic supported by the application. From a performance perspective, an API gateway provides several benefits, such as options to set up request authentication, rate-limiting to prevent service over-use and DDoS attacks, monitoring and analytics endpoints (using Spring Boot Actuator), reducing latency by serving cached responses, as well as load balancing. A very simple gateway configuration example is shown below. The `lb://` prefix in `uri:` makes use of Spring Cloud Gateway's built-in load balancer, to provide round robin client-side load balancing features during calls to other microservices. A 503 Service Unavailable Error is returned when a service cannot be found, which is more appropriate than a basic Java exception stack trace shown when using `http://` or `https://` prefixes instead.

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: myRoute
        uri: lb://service
        predicates:
        - Path=/service/**
```

Listing 5.1: Sample Spring Cloud Gateway configuration

Spring Cloud Gateway paves the way for discussion on the next two design patterns: *circuit breaker* and *service discovery*, thanks to the Spring integrations available to simplify configuration and maintenance.

[1] https://spring.io/projects/spring-cloud
[2] https://spring.io/projects/spring-cloud-gateway
[3] https://spring.io/projects/spring-boot

Circuit Breaker

In order to prevent cascading network or service failures resulting in system performance, the *circuit breaker* pattern is best applied in the API gateway service to adopt a "fail-fast" approach. This is primarily achieved by adding a `CircuitBreaker` filter to the Spring Cloud API gateway routing logic, a snippet of which is shown below.

```
spring:
  application:
    name: api-gateway-service
  cloud:
    gateway:
      routes:
        - id: aggregator-service-route
          uri: lb://AGGREGATOR-SERVICE
          predicates:
            - Path=/api/**
          filters:
            - name: CircuitBreaker
              args:
                name: aggregator-service-cb
                fallbackUri: forward:/fallback/aggregator-service
        - id: cinema-cineworld-service-route
          uri: lb://CINEMA-CINEWORLD-SERVICE
          predicates:
            - Path=/cinema/cineworld/**
          filters:
            - name: CircuitBreaker
              args:
                name: cinema-cineworld-service-cb
                fallbackUri: forward:/fallback/cinema-cineworld-service
```

Listing 5.2: Snippet from the API gateway service's application properties

Spring Cloud supports a number of different circuit breaker implementations such as Netflix Hystrix, Resilience4J, Sentinel and Spring Retry, of which reactive Resilience4J [4] is preferred when used together with Spring Cloud Gateway as shown in the listing below. Reactivity is necessary in the circuit breaker since Spring Cloud Gateway is itself built using Project Reactor [5].

```
private Resilience4JConfigBuilder builder(String id) {
  return new Resilience4JConfigBuilder(id)
          .timeLimiterConfig(
                  TimeLimiterConfig.custom().timeoutDuration(Duration.
  ofSeconds(3)).build())
          .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults());
}
```

Listing 5.3: Circuit breaker configuration in API gateway application

In the API gateway application, Resilience4J is configured with default settings, plus a timeout of *3 seconds* for demonstration purposes. This implies that any API endpoint accessed via the gateway that takes over 3 seconds to respond will cause the circuit breaker to trip, and forward the requester to the (specified for every route in the gateway configuration application properties).

[4] https://github.com/resilience4j/resilience4j
[5] https://projectreactor.io/docs

The timeout ensures that no transient faults in a microservice will degrade the system performance (primarily response time).

```java
@GetMapping("/{name}")
public String message(@PathVariable("name") String serviceName) {
    return String.format("Fallback: Circuit broken in %s!", serviceName);
}
```

Listing 5.4: Code snippet from `FallbackController.java`

The fallback endpoint is shown above, and in this application it simply logs the name of the service where the circuit was broken.

Service Discovery

Service discovery is an essential design pattern for modern microservices, where the network location (host name/IP address and port number) may be dynamic due to the industry dominance of virtualisation and containerisation solutions to run microservice-based applications. Hard-coding the locations of services is then considered an anti-pattern, since it necessitates changes in multiple places when services are scaled up or down. Hence, a natural solution is to set up a service registry, which maintains the locations of all active services, and provides access to registered services through logical addresses (such as `lb://AGGREGATOR-SERVICE` or `lb://CINEMA-CINEWORLD-SERVICE` instead of actual network locations).

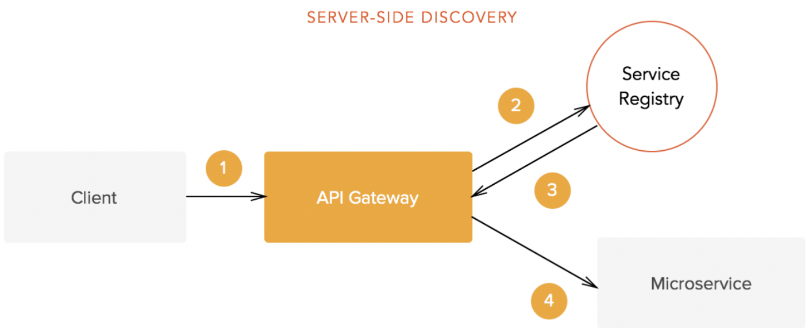

Figure 5.2: A simple service registry setup.
Source: [41]

Fig. 5.2 above depicts the gist of the service discovery pattern, and Fig. 5.1 shows how a service registry is implemented in this case study using the Spring Cloud Netflix (Eureka) project [6]. The server and client Maven dependencies are `spring-cloud-starter-netflix-eureka-server` and `spring-cloud-starter-netflix-eureka-client` respectively. The `eureka-server` service uses an `@EnableEurekaServer` annotation in its main application class to denote its server status (maintaining the service registry), and every other microservice uses a corresponding `@EnableEurekaClient` annotation (to self-register with the Eureka server on application startup). The server and sample client Configuration are shown in the following listings:

```yaml
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

Listing 5.5: Snippet from Eureka server's application properties

[6] https://spring.io/projects/spring-cloud-netflix

The server is prevented from trying to register itself. For the sample client configuration, the Spring Boot application attempts to contact the Eureka server on `code.client.serviceUrl.defaultZone`, which is specified in `docker-compose.yml`. Eureka clients also send regular heartbeat messages to the server (set using `lease-renewal-interval-in-seconds`) to maintain their active status. Once the server stops receiving heartbeats below an expected threshold, it starts evicting the instances from the registry; this is known as self-preservation.

```
eureka:
  instance:
    lease-renewal-interval-in-seconds: 5
  client:
    register-with-eureka: true
    fetch-registry: true
    serviceUrl:
      defaultZone: http://${EUREKA_SERVER:eureka-server}:${EUREKA_PORT
  :8761}/eureka/
```

Listing 5.6: Snippet from a Eureka client's application properties

Netflix Eureka also makes a dashboard available (Fig. 5.3) at the configured port (8761), which shows the active services, system status, and other useful information about memory usage, number of CPUs, server uptime, and so on.

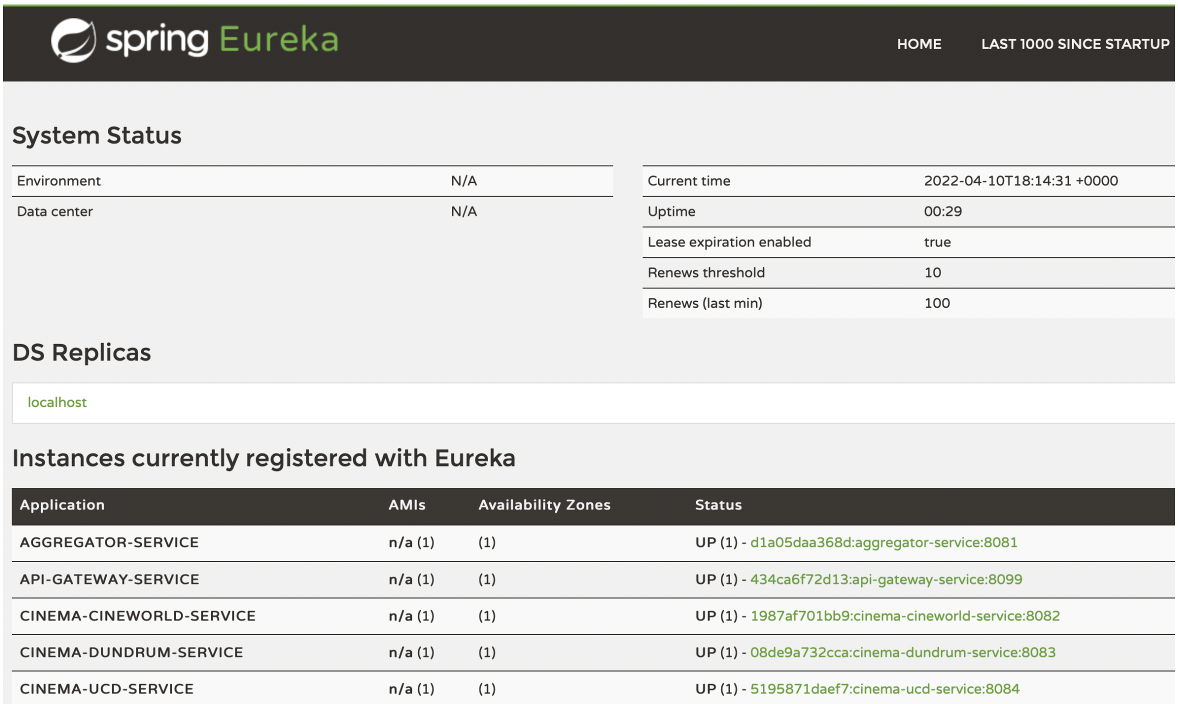

Figure 5.3: Eureka dashboard

Aggregator

The service *aggregator* design pattern for microservices is useful when similar calls need to made to multiple microservices. On receiving a request from the client via an API gateway, the aggregator service dispatches the request to appropriate internal backend microservices, then combines the responses into a single structure, which is returned to the requester. The aggregator itself doesn't need to know the routing details of internal services, since it can communicate with them through the API gateway again. The performance benefit of this pattern is the reduction in communication overhead and chattiness between the client and the various internal cinema microservices.

Moreover, in this case study, the aggregator service relies on the service discovery pattern to find the registered cinema services at any given time. The code snippet from `aggregator-service` below illustrates how a custom discovery service function is used to fetch the addresses of all active cinemas, which the aggregator can use to send identical GET or POST requests to (received from client via the gateway). The primary advantage of this design choice is the fact that the aggregator does not need to be made aware when the number of cinema services changes, as long as the necessary registration process is completed with the Eureka server.

```java
for (String serviceUrl : discoveryService.getCinemaUrlPrefixes()) {
    String url = apiGatewayHost + serviceUrl + endpoint;
    if (httpMethod.matches("GET")) {
        results.add(restClient.getForObject(url, type));
    } else if (httpMethod.matches("POST")) {
        results.add(restClient.postForObject(url, body, type));
    }
}
```

Listing 5.7: Snippet from `AggregatorService.java`

In the given movie ticket reservation system, the function of the aggregator is to gather the list of movie showtimes from every registered cinema, and return a compiled list to the client. It should also be noted that an alternative to an aggregator microservice is to perform response aggregation in an API gateway itself, which results in the *gateway aggregation* pattern.

Externalised Configuration

Externalised configuration is a cross-cutting concerns design pattern, that is especially useful in separating changeable application properties from the business logic. Spring Boot has built-in support for processing external configuration from various sources, such as Java properties files, YAML files, environment variables, and command-line arguments. The values set by the developer for such properties are injected into Spring beans [7] using the `@Value` annotation (among other alternatives).

Shown below are the environment variable settings for Cineworld's cinema service in the first case study's Docker Compose file.

```yaml
cinema-cineworld-service:
  ...
  environment:
    - SERVER_PORT=8082
    - EUREKA_SERVER=eureka-server
    - DATABASE_HOST=mongodb
    - DATABASE_PORT=27017
  ...
```

Listing 5.8: Snippet from `docker-compose.yml`

Using the above environment variables, Spring Boot then sets the application properties. Externalising configuration also allows the development team to maintain all properties in one place (say the Docker Compose file) such that any required changes will only need to be made in that file, instead of searching for every occurrence of the property in the application source code.

```yaml
server:
  servlet:
    context-path: /cinema/cineworld
```

[7] https://www.baeldung.com/spring-bean

```
  port: ${SERVER_PORT}
spring:
  application:
    name: cinema-cineworld-service
  data:
    mongodb:
      database: cinema-cineworld
      host: ${DATABASE_HOST:mongodb}:${DATABASE_PORT:27017}
eureka:
  ...
  client:
    ...
    serviceUrl:
      defaultZone: http://${EUREKA_SERVER:eureka-server}:${EUREKA_PORT
  :8761}/eureka/
```

Listing 5.9: Snippet from `cinema-cineworld-service` `application.yml` file

Database per Service

In monolithic applications, databases are shared across services, typically in a tiered approach (involving a web tier, services tier, cache tier and data tier). However, this leads to a large central database that becomes a single performance bottleneck for all data operations, and is considered an anti-pattern when used with microservice architecture. For the microservices in our system to be truly independent, their persistent data stores must also be loosely coupled. Each service must have ownership over its domain data and logic under an autonomous lifecycle. This is achieved using the *database per service* design pattern.

Database independence in microservices can possibly introduce additional concerns, as illustrated by the CAP theorem [8], according to which a distributed data store can only provide two of the following: consistency, availability and partition tolerance (see also: choice between ACID and BASE transaction models [9]). In this project, MongoDB [10], a popular NoSQL database, is set up using Docker.

```
services:
  mongodb:
    image: mongo:latest
    container_name: mongodb
```

Listing 5.10: Docker Compose file snippet for MongoDB server

Although only a single server is used in this project (serving at the default port - 27017), every microservice is configured to use a separate database. For instance, in the code listing below, `spring.data.mongodb.database` is set to `cinema-cineworld`, which creates and uses a separate database for Cineworld service, independent from other cinemas. When dealing with services with high throughput, multiple database servers may need to be provisioned, which can be achieved with Docker service scaling or using an orchestration tool like Kubernetes. The complications of a web application's integration with a persistent data store is made trivial with the help of the Spring Data MongoDB [11] project.

```
spring:
```

[8] https://en.wikipedia.org/wiki/CAP_theorem
[9] https://www.geeksforgeeks.org/acid-model-vs-base-model-for-database/
[10] https://www.mongodb.com
[11] https://spring.io/projects/spring-data-mongodb

```
  ...
  data:
    mongodb:
      database: cinema-cineworld
      host: ${DATABASE_HOST:mongodb}:${DATABASE_PORT:27017}
```

Listing 5.11: Snippet from Cineworld cinema's application properties

Although all services use MongoDB in this project for the purpose of simplicity, the database per service design pattern allows every service to use a different kind of data store if the need arises, for instance, a service storing a lot of relational data could use MySQL, whereas another service storing network data might use a suitable graph database.

Service Instance per Container

As discussed in the project outline, Docker is used to containerise and deploy the microservices. The services have been built as a multi-module Maven project, each having an independent Dockerfile for deployment. Docker Compose [12] simplifies the process of defining and running multi-container Docker applications: running `docker-compose up` starts and runs the services defined in the Compose file, provided that each service has a valid Dockerfile.

```
docker-compose down --remove-orphans
docker-compose build --no-cache
docker-compose up
```

Listing 5.12: Sample Docker Compose commands to start the microservices

The wide range of benefits to containerising microservices make Docker the industry leading tool for service deployment. Developers are able to choose any suitable programming language and framework (including different versions), since Docker provides isolated environments with all required dependencies. The *service instance per container* design pattern allows each microservice to be packaged as a Docker container image, permitting the deployment of multiple instances of the microservice as separate containers if required.

There are far reaching performance benefits of this pattern, including service reproducibility, independent deployment and scaling, hardware resource (CPU/memory) constraining capabilities, service instance monitoring, reliability and fault tolerance. Most importantly, Docker containers are much faster than traditional virtual machines (VMs) thanks to lightweight architecture (sharing the host OS kernel instead of requiring a complete OS or hypervisor). In large scale projects, container orchestration tools such as Kubernetes [13], Marathon ( ) [14], Amazon ECS [15] and EKS [16] are widely used to automate the operational effort required to manage containerised microservices, which is key for well-run DevOps.

The three cinema microservices have identical Dockerfiles, as shown below. Each of them use a JDK 11 base image, copy the compiled JAR file into the container, then run it after a 5 second wait. Using Docker Compose, each service's Dockerfile is used to start the application.

```
FROM openjdk:11-jre-slim
COPY target/cinema*.jar /cinema.jar
CMD sleep 5 && java -jar /cinema.jar
```

Listing 5.13: Dockerfile for cinema services

[12] https://docs.docker.com/compose/
[13] https://kubernetes.io
[14] https://mesosphere.github.io/marathon/
[15] https://aws.amazon.com/ecs/
[16] https://aws.amazon.com/eks/

5.1.2 Evaluation

5.2 Case Study 2

5.2.1 Design and Implementation

Asynchronous Messaging

Application Metrics

Externalised Configuration

Database per Service

Service Instance per Container

5.2.2 Evaluation

Chapter 6: Conclusions

TODO: Include: key takeaways from experiments and performance analysis, future work and improvements, reflection (lessons learnt)

6.1 Summary

6.2 Future Work

6.3 Reflection

TODO

In this section you will sum up your report, draw some conclusions about your work so far, and make some general observations about the work to come. You may also use this opportunity to express points of view, or make factual claims, that are more pertinent here than in other sections of the report. If your project raises some ethical concerns, for example about how data or users are treated, then address them here in a thoughtful manner.

Regarding this document, here are some concluding points that you should keep in mind when writing your own. You may use screenshots in your report, but do not overfill your report with them, or with figures of any kind. Make sure that figures earn their keep, and are not just present as space fillers or as eye candy. If you use diagrams or figures from other people's work, including the web, be sure to cite the creator in the corresponding caption. All things being equal, it is better to construct your own figures than to copy and paste those of others. In any case, always make sure that your images are readable, do not suffer from pixelation or aliasing effects, and that each is clearly numbered, captioned and meaningfully referenced in the main body of the text.

Ensure that there is a cohesive argument expressed in the text of the report and that it is not simply a bag of diagrams, screenshots and wishful thinking. Every report should tell a story, so know what story you want to tell. When you include images, make sure they are readable and truly add to the discussion.

Make sure your language is professional throughout, and steer a course between pompous and colloquial. Maintain authorial distance and do not overuse "me," "I" and "our." Your are writing for a professional audience who will judge you on the quality of your prose, so use a grammar and a spelling checker.

Use LaTeX if you wish - this is recommended if you plan to use mathematical formulae in your report, but in any case, keep the general spacing and font/style you find here (Single or 1.5 spacing, 12 pt. font for text, etc.). Be sure to submit a PDF (never a .DOC or .DOCX file) as your report. If you prepare your report in MS Word, as this document has been, save it as a PDF before you submit it. Overall it should be about 18 - 20 pages, including figures, front matter and references, A significant portion of the report will be textual, with approx.. five or six thousand words. Do not rely on images or other filler to write your report for you. The dates and means of submission will be communicated to you separately.

Acknowledgements

I would like to express my sincere gratitude towards:

- Professor John Murphy, my supervisor, for his guidance, support and understanding throughout the project.
- My team at Amazon Web Services (Dublin), and Associate Professor Rem Collier (UCD), for introducing me to the world of distributed systems and microservices.
- UCD School of Computer Science, for providing access to a Linux compute server.
- My friends and family, for their constant encouragement.

Bibliography

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

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

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

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

[5] M. Kamaruzzaman. 'Effective Microservices: 10 Best Practices.' (23rd Nov. 2019), [Online]. Available: https://t.co/ZM78yg19OR?amp=1 (visited on 26th Oct. 2021).

[6] M. Kamaruzzaman. 'Microservice Architecture and its 10 Most Important Design Patterns.' (15th Dec. 2020), [Online]. Available: https://t.co/9Du1U3X4Q6?amp=1 (visited on 26th Oct. 2021).

[7] S. Kappagantula. 'Everything You Need To Know About Microservices Design Patterns.' (25th Nov. 2020), [Online]. Available: https://www.edureka.co/blog/microservices-design-patterns (visited on 26th Oct. 2021).

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

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

[10] K. Cully, 'Performance Problems Inherent to Microservices with Independent Communication and Resiliency Configuration,' M.S. thesis, University College Dublin, Ireland, Mar. 2020.

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

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

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

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

[15] C. Pahl and P. Jamshidi, 'Microservices: A Systematic Mapping Study,' Jan. 2016, pp. 137–146. DOI: 10.5220/0005785501370146.

[16] N. Alshuqayran, N. Ali and R. Evans, 'A Systematic Mapping Study in Microservice Architecture,' Nov. 2016, pp. 44–51. DOI: 10.1109/SOCA.2016.15.

[17] P. Di Francesco, P. Lago and I. Malavolta, 'Architecting with Microservices: a Systematic Mapping Study,' *Journal of Systems and Software*, vol. 150, Apr. 2019. DOI: 10.1016/j.jss.2019.01.001.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[33] A. O. Portillo-Dominguez, M. Wang, J. Murphy, D. Magoni, N. Mitchell, P. F. Sweeney and E. Altman, 'Towards an Automated Approach to Use Expert Systems in the Performance Testing of Distributed Systems,' New York, NY, USA: Association for Computing Machinery, 2014. DOI: 10.1145/2631890.2631895.

[34] O. Portillo, P. Perry, D. Magoni and J. Murphy, 'PHOEBE: an automation framework for the effective usage of diagnosis tools in the performance testing of clustered systems,' *Software: Practice and Experience*, Apr. 2017. DOI: 10.1002/spe.2500.

[35] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, 'Performance Evaluation of Microservices Architectures Using Containers,' in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34. DOI: 10.1109/NCA.2015.49.

[36] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte and J. Wettinger, 'Performance Engineering for Microservices: Research Challenges and Directions,' in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ser. ICPE '17 Companion, L'Aquila, Italy: Association for Computing Machinery, 2017, pp. 223–226, ISBN: 9781450348997. DOI: 10.1145/3053600.3053653.

[37] M. Gribaudo, M. Iacono and D. Manini, 'Performance Evaluation Of Massively Distributed Microservices Based Applications,' May 2017, pp. 598–604. DOI: 10.7148/2017-0598.

[38] S. Eismann, C.-P. Bezemer, W. Shang, D. Okanović and A. van Hoorn, 'Microservices: A Performance Tester's Dream or Nightmare?' In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 138–149, ISBN: 9781450369916. DOI: 10.1145/3358960.3379124.

[39] A. U. Gias, A. van Hoorn, L. Zhu, G. Casale, T. F. Düllmann and M. Wurster, 'Performance Engineering for Microservices and Serverless Applications: The RADON Approach,' in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 46–49, ISBN: 9781450371094. DOI: 10.1145/3375555.3383120.

[40] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes and J. Saraiva, 'Energy efficiency across programming languages: How do energy, time, and memory relate?' In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267, ISBN: 9781450355254. DOI: 10.1145/3136014.3136031.

[41] S. Peyrott. 'An Introduction to Microservices, Part 3: The Service Registry.' (2nd Oct. 2015), [Online]. Available: https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/ (visited on 10th Apr. 2022).

Chapter 7: Appendix

7.1 Code Repository

The code for this project can be found on GitLab (internal UCD server) and GitHub (eventually):

https://csgitlab.ucd.ie/rajitbanerjee/microservice-design-patterns
https://github.com/rajitbanerjee/microservice-design-patterns

7.2 Project Work Plan

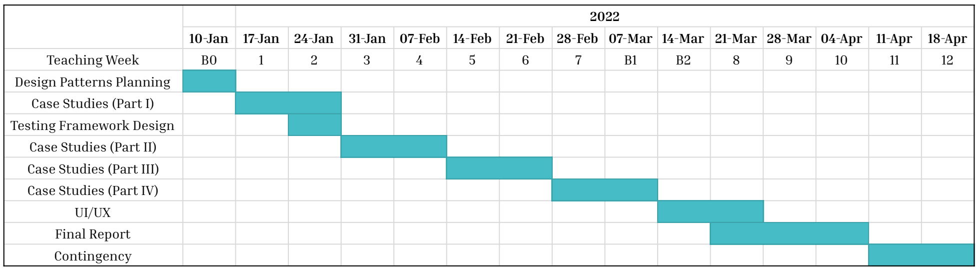

Figure 7.1: Gantt chart for project timeline

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