UNIVERSITY OF CALIFORNIA,
IRVINE

Microservice Pattern Identification from Recovered Architectures of Orchestrated Systems

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Aaron P. Matthews

Thesis Committee:
Assistant Professor Joshua Garcia, Chair
Professor André van der Hoek
Professor Sam Malek

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My thanks to Dr. Garcia for the guidance on my thesis,
My thanks to Dr. George for the computer engineering research opportunities at CSUF,
My thanks to Dr. Clahane for the mathematics research opportunities at Fullerton College,
My thanks to my family for the enduring support.

ABSTRACT OF THE THESIS

Microservice Pattern Identification from Recovered Architectures of Orchestrated Systems

By

Aaron P. Matthews

Master of Science in Software Engineering

University of California, Irvine, 2021

Assistant Professor Joshua Garcia, Chair

Microservice architecture has become widely-used in industry, with tech giants like Amazon, Twitter, and LinkedIn leveraging microservices to evolve their web-scale applications. Microservice architecture brings benefits such as scalability and technological heterogeneity, although at a cost of complexity. A number of microservice patterns have been proposed to address this cost. This thesis explores the use of microservice patterns in recovered microservice architectures; however, there have been few attempts at recoveries of microservice systems. The contributions of this thesis are: 1. A definition of microservices as an architectural style, 2. A recovery technique for orchestrated systems which has been applied to three benchmark systems, and 3. A microservice pattern identification technique for five patterns, and applying it to the three recovered systems. In our results, 4 of the 5 patterns in all three benchmark systems were successfully identified.

Chapter 1

Introduction

The microservice architecture style has become widely-used in industry, particularly among web-scale products like those developed at Amazon, Netflix, Twitter, and LinkedIn [1]. The style has also found applications in emerging technologies such as IoT and Fog computing [2]. According to Allied Research, microservice architecture was valued at $2,037 million in 2018, and is projected to reach $8,073 million by 2026 [3].

While there have been many proposed techniques on the architecture recovery of object-oriented systems [4], there have been few attempts at the recovery of microservice systems [5] [6]. This paper proposes a recovery technique which can be applied to Kubernetes-orchestrated microservice systems and applies it to three microservice benchmark systems.

To address the complexities of microservice architecture, both the research community and industry have proposed microservice patterns to ameliorate common problems for such systems. There are documented patterns to ease issues of Communication, Design, Orchestration, Migration, et cetera [7]. While there have been some attempts to identify anti-patterns, i.e., smells [8], in microservice systems, there have been almost no attempts to identify patterns. In this paper, we analyze recovered architectures to identify microservice patterns

which can emerge at an architectural level.

The motivation for microservice pattern identification is threefold: One, techniques for identification can aid researchers in mining usage of microservice patterns. Two, pattern identification tools can be used for auto-documentation to provide information to developers who are learning a system. Three, pattern identification can be used to enforce architectural constraints, e.g., in a CI pipline to report on pattern usage.

The thesis structure is as follows: in Chapter 2, microservice architecture is presented as an architectural style. In Chapter 3 the architectural recovery technique is described. In Chapter 4, the microservice pattern identification technique is described. In Chapter 5, results of the recovery and pattern identification are presented and discussed. Related work is discussed in Chapter 6, and the thesis is concluded in Chapter 7.

Chapter 2

Microservice Architectural Style

Taylor et al. offer a rubric to define architectural styles [9]. This rubric is applied to the microservice style below:

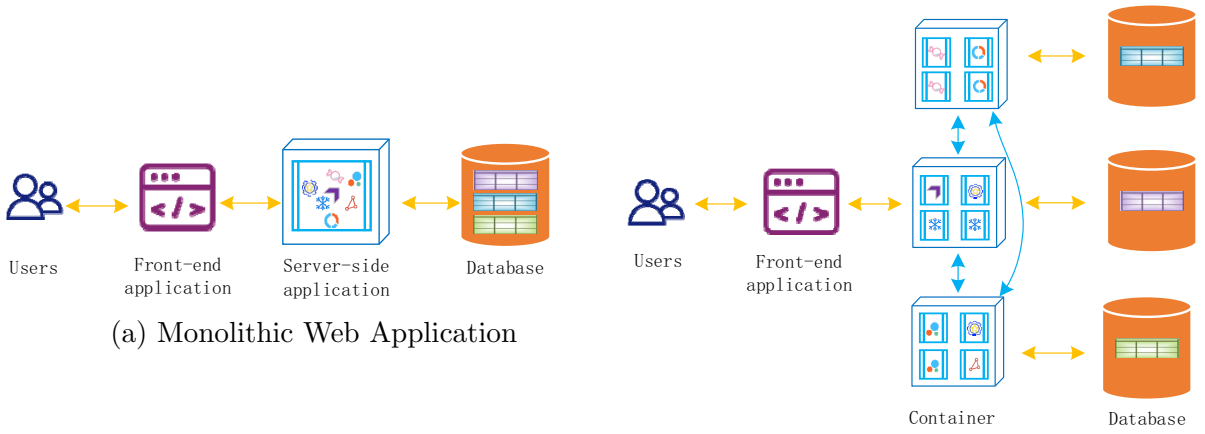
- *Summary* – An application is divided into loosely-coupled, highly-cohesive, technologically heterogeneous services with a limited range of features, which communicate using light-weight mechanisms.
- *Components* – Small services built around business capabilities. May include utility components for web servers, message brokerage, service discovery, centralized logging and/or monitoring.
- *Connectors* – Light-weight communication mechanisms such as RESTful APIs, RPC frameworks and/or message brokers.
- *Data Elements* – Documents typically using JSON interchange or a serialized RPC format.
- *Topology* – Dependencies between microservices using REST or RPC will form a directed graph topology. If an event bus is used for all communication, this will elicit

a hub-and-spoke topology. Microservice systems may use a combination of both communication styles.

- *Additional Constraints* – Microservices should have strong service boundaries, they should be independently deployable, and the services should be small.
- *Variants* – FaaS (Function as a Service), e.g., AWS Lambda are similar to microservices, but with very narrow boundaries like that of a function.
- *Qualities Yielded*
  - *Design*: Microservice designs will have greater modularity, lowering technical debt [10].
  - *Technology*: The most appropriate technology, i.e., language / stack, can be selected for each microservice.
  - *DevOps*: Individually deployable microservices can result in more frequent deployments and reduce downtime due to fewer single points of failure [11].
  - *Teamwork*: Independent teams can develop faster and grow more easily [12].
  - *Infrastructure*: Microservice systems can scale more efficiently, reducing infrastructure costs [13].
- *Typical Uses* – Complex web-scale applications.
- *Cautions*
  - *Design*: How small a microservice should be presents a design challenge, as there "is no definition of how small a microservice should be" [12].
  - *Technology*: Greater complexity due to being a distributed system and trade-offs for data consistency [11].
  - *Teamwork*: More difficulties in transferring team members due to technological heterogeneity [12].

– *Infrastructure*: There is a need for sophisticated levels of automation and monitoring strategies [12].

An illustration of a monolithic versus microservice system from Liu et. al [1] can be seen in Figures 2.1a and 2.1b. In Figure 2.1a, we see a large server-side application which handles many domain concerns, whereas in Figure 2.1b we see this large application has been decomposed into smaller components (i.e., microservices), each of which handle a subset of the concerns for the domain. The connectors between the components (light blue) represent light-weight communication mechanisms, such as REST APIs.



(a) Monolithic Web Application

(b) Microservice Web Application

Figure 2.1: Monolithic vs. Microservice Architecture

Chapter 3

Architecture Recovery

Existing architecture recovery techniques are typically language-dependent, often only supporting recovery of Java programs. Microservice architecture recovery presents a challenge due to:

- Microservice systems are often polyglot (i.e., written with multiple programming languages), making static analysis of the entire system difficult.
- Microservice components communicate over *ambiguous interfaces*, wherein there are general entry points into a component [14]. Ambiguous interfaces are an architectural smell as they undermine static analysis. Microservices, which communicate with each other via message brokers or RESTful interfaces will, yield few opportunities for static analysis.

Since technological heterogeneity creates a challenge, what is technologically homogeneous about microservice systems that can be leveraged for architectural recovery? Two possibilities are *containerization*, and *orchestration*.

3.1 Containerization

Containerization allows encapsulation of software applications and their dependencies. It provides the same advantages of virtualisation, but with more efficient resource utilization [15]. This is due to the fact that containers share operating system resources, e.g., the kernel, whilst virtual machines host a complete operating system. This efficient encapsulation makes containers ideal for the deployment of a microservice.

The most widely-used containerization tool is Docker. A docker container is built from a simple key-value configuration file known as a `Dockerfile`. Once built, the resulting container is known as an *image*. Docker containers are hierarchical, in the `Dockerfile` the key `FROM` indicates which image the container is based on. The images are pulled from an image repository, the most popular being *Dockerhub* at `hub.docker.com`. This repository is host to millions of images, with the most popular images having 10s of millions of downloads.

Given the popularity of Docker and Dockerhub, the image name in the `FROM` key provides insight into what software the container holds. Images names, e.g., `nginx` or `httpd` (Apache) would indicate a container for web servers. Image names, e.g., `redis` or `memcached` would indicate a container for caching. Image names, e.g., `mysql` or `postgres` would indicate a relational database for peristence. And image names, e.g., `node`, `python`, or `openjdk` would indicate some domain-specific software in its respective language.

Figure 3.1 shows an example Dockerfile which creates a Java 8 runtime environment, copies a `.jar` file into the container, launches the JAR and exposes an application port to the container engine.

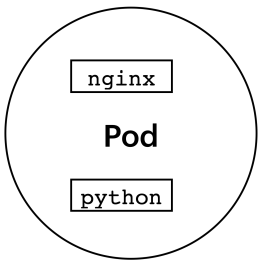
```
FROM java:8-jre
ADD ./target/ts-auth-service-1.0.jar /app/
CMD ["java", "-Xmx200m", "-jar", "/app/ts-auth-service-1.0.jar"]
EXPOSE 12349
```

Figure 3.1: Example Dockerfile

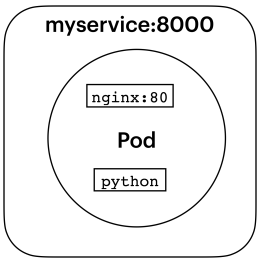
3.2 Container Orchestration

Microservice systems are deployed with 10s, 100s, or 1000s of containers. At large scale, it is not feasible to manage the deployed containers via manual means. Orchestration automates many aspects of the deployment and management of containers. Container Orchestration can provide automated deployment strategies such as canary releases, where new deployments are released to a gradually-increasing subset of users. Orchestration can also manage container replicas based on policy, and handle fault-tolerance tasks when containers fail.

Kubernetes, also referred to as K8s, is a widely-used container orchestration tool developed by Google and released in 2014. Kubernetes has been shown to outperform other orchestration tools [16] and is frequently used in microservice benchmark systems. The atomic unit in Kubernetes is a *pod*, which is a set of Docker containers intended to function together. For example, we may have a pod with a `python` container hosting python code for a web service, and an `nginx` container to serve the python endpoints over HTTP, see Figure 3.2a.



(a) Illustration of a Kubernetes Pod, containing a `nginx` webserver image and a `python` runtime environment image.



(b) Illustration of Kubernetes Service, exposing an `nginx` webserver.

Figure 3.2

In order to expose pods to the outside, a *service* is defined in Kubernetes. This service exposes an external network port which forwards to an internal network port on a pod. In our aforementioned example, an external port would forward to the internal HTTP port 80 handled by the `nginx` container in the pod. The other function provided by a service is the management of replicas, in each service a number of replicas is defined. Kubernetes will attempt to keep the number of replicas at the defined limit by spawning new replicas as pods fail. The service also acts as a load balancer, choosing pods to handle incoming connections as they arrive at the service's exposed port. An example is seen in Figure 3.2b.

One last functionality provided by services is service discovery. Service discovery is provided via DNS. When a service is defined, a name is configured for the service. Kubernetes exposes the service names to the cluster in response to DNS requests for the service name. A Kubernetes service is the ideal way to manage a microservice – exposing its needed ports, load balancing its replicas, and leveraging service discovery to be found by other microservices.

3.3 Methodology

To recover the architecture, a dynamic analysis technique will be used, i.e., architectural information will be extract during the execution of the microservice system. The implementation of this technique uses Python as its language. At a high level, there are four steps: A. Query K8s for services along with their network & container details, B. Start capturing network traffic within the system, C. Execute domain use cases, and D. Analyze captured traffic to find dependencies between microservices. The steps are detailed below.

3.3.1 Query K8s for services along with their network & container details

The Python library `kubernetes` is used to communicate with the K8s cluster. When querying the cluster for running services, the library returns for each service its DNS name, IP address, exposed port and forwarded internal port. When querying the cluster for running pods, the library returns for each pod its IP address and container image name, e.g., `mongo`. Using this image name, a mapping takes place to categorize the pod as either a database, a message broker, or a service. The mapping table is in Table 3.1. This mapping is easily extendable however this set was sufficient for the chosen benchmark systems.

Database Images	Broker Images
`mongo`	`rabbitmq`
`mysql`	`*/activemq`
`redis`	`*/kafka`
`*/mssql`	
`postgres`	

Table 3.1: Image names associated with service types

If the service type cannot be inferred from the image name, as a fallback, it is determined from the internal port of the pod. Common ports such as 27017 and 3306 indicate a database service for mongodb and mysql, respectively.

3.3.2 Start capturing network traffic within the system

Kubernetes creates virtual network devices for the services to communicate with one another. The network provider in our cluster was Calico, and all the vnet devices start with "cali". The Python library `scapy` is used to query the operating system for available network devices and the devices with the aforementioned prefix are passed to `scapy`'s packet sniffer.

The packet sniffer works similarly to network analysis tools like Wireshark or Tcpdump.

As packets are captured on the network devices, the source & destination IP and source & destination port are recorded.

3.3.3 Execute domain use cases

In this step, use cases which elicit network communication between the components are executed. These use cases may come from the system's integration tests or load tests. If no tests are available, manually scripted use cases are used. If manually scripted use cases are necessary, they are typically derived from the documentation and exploratory testing. Use cases are found and scripted until the recovered architecture does not yield any new components or connectors. We'll discuss the use cases for indivudal test subjects in Chapter 5.

3.3.4 Analyze captured traffic to find dependencies between services

Recall that a service may have multiple replicas of a pod. Each service is assigned an IP address and each of its pods is assigned an IP address. Using the information we queried in Section 3.3.1, we can map each captured packet's source and destination IP address to its corresponding service. Packets which do not originate from or arrive to a service are ignored; these are typically packets for DNS lookups.

The services (such as microservices with REST APIs) communicate as a client-server relationship. In this scenario a client has a *dependency* on a server. To identify dependenceies, the packets we are interested in are those in which a client initiates a request to a server. The server (e.g., a microservice or a database) will respond to the request and return data back to the client, but these packets can be ignored. To determine that a client is initiating

a request to a server, the destination port is examined, and if the destination port of the packet is equal to the K8s service's external port, then a dependency is noted between the client's service and the server's service.

For example, consider a client pod which depends on a database, as seen in Figure 3.3. When the pod initiates a connection to the *Database Service*, it connects to the service's external port, in this case port 9000. The service, acting as a load balancer, forwards the request to one of its replicated database pods. The chosen database pod will respond to the client on the random port opened by the client to receive the response, in this case port 12345. This response traffic is ignored in the analysis, as well as the traffic between the service and the chosen pod. Only the initial connection from the client pod to the database service on its external port is used to identify the dependency.

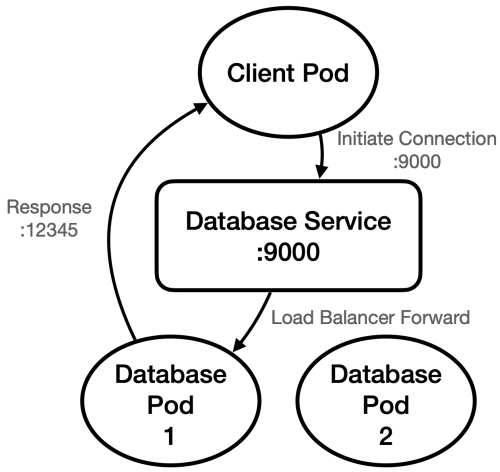


Figure 3.3: Example Traffic between Client Pod and K8s Service

The output of this step is a directed graph, with the services as the vertices and the dependencies as the edges. The graph is exported as a GraphViz `.gv` file for rendering and a `.csv` file for importing into Neo4j.

Chapter 4

Microservice Patterns and Identification

Osses et. al catalogued and categorized over 100 microservices patterns, from both academic and industry sources [7]. Each pattern's source documentation was reviewed to determine if it was likely to:

1. Be present at an architectural level, and
2. Be identifiable via automated means.

Some patterns are inherently present in an orchestrated microservice system, for example the *Containerize the Services* pattern and *Introduce Service Discovery* deployment patterns are necessarily present when using Kubernetes. These trivially identified patterns were omitted. The remaining patterns are described in Section 4.2.

4.1 Methodology

The output of the recovery technique in Chapter 3 is a directed graph, where each vertex is a K8s service and each edge represents a dependency between two services. Each vertex has a type, which can be one of a *Database*, a *Broker*, or a *Service*. The analysis tool outputs all this information in a `.csv` file.

The `.csv` file is imported into Neo4j, a popular graph database system [17] for analysis. Neo4j uses a query language similar to SQL but for graph structures. The query language, Cypher, is the basis for an open graph query standard in development by ISO [18].

Cypher queries are devised to identify sub-graphs or relationships that potentially identify a microservice pattern. The query results from Neo4j will report the presence of a pattern and also the number of instances of said pattern. The queries for the patterns are described in the following section.

4.2 The Patterns

4.2.1 Shared Database Pattern

The shared database pattern allows multiple microservices to use the same database [19]. This can be necessary for microservices that require a high degree of data consistency, i.e., there is little tolerance for eventual-consistency schemes. The disadvantage to this pattern is that it creates coupling between microservices which share the same database.

This pattern can be identified when there is more than one service component which depends on the same database component, see Figure 4.1 for an entity-relationship. The Cypher query can be seen in Figure 4.2.

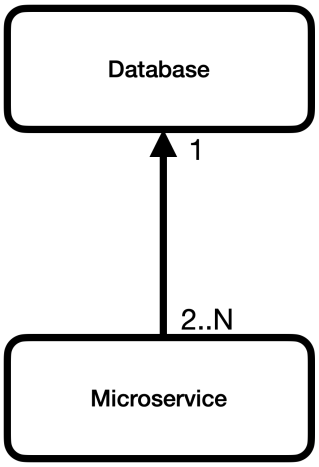


Figure 4.1: Shared Database Component Relationship

```
MATCH (d:Database)<-[:DEPENDS_ON]-(s:Service)
WITH d,count(distinct s) AS c
WHERE c > 1 RETURN d.name,c
```

Figure 4.2: Shared Database Query

4.2.2 Database-per-Service Pattern

Opposite of the Shared Database pattern is the Database-per-Service pattern, which dictates that no two microservices may share a database [19]. The advantage of this approach is the decoupling achieved between microservices. A drawback is less consistency between microservices which use the same data, and the complexity of implementing data transactions.

This pattern can be identified when there is only a single service dependency on each database component, see Figure 4.3 for an entity-relationship. The Cypher query can be seen in Figure 4.4.

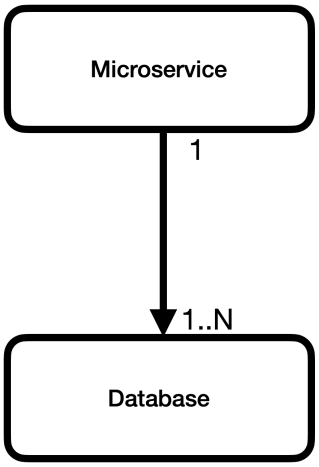


Figure 4.3: Database-per-Service Component Relationship

```
MATCH (d:Database)<-[:DEPENDS_ON]-(s:Service)
WITH d,count(distinct s) AS c
WHERE c = 1 RETURN d.name,c
```

Figure 4.4: Database-per-Service Query

4.2.3 Asynchronous Messaging Pattern

Microservices typically need to communicate with one another, and they can communicate directly by calling each other's REST APIs. However this simple approach leads to a coupling between the microservices. The Asynchronous Messaging pattern decouples the communication between microservices by using a message broker [19]. The disadvantage of this approach is the architectural smell of ambiguous interfaces [4].

This pattern can be identified when there is a message broker which has multiple microservices that depend on it, see Figure 4.5 for an entity-relationship. The Cypher query can be seen in Figure 4.6.

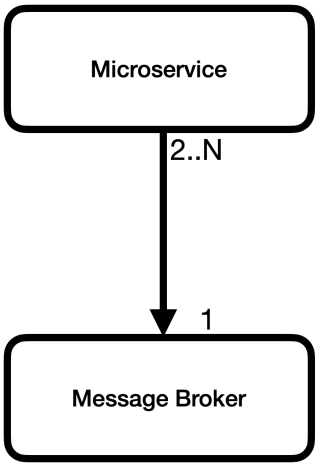


Figure 4.5: Asynchronous Messaging Component Relationship

```
MATCH (b:Broker)<-[:DEPENDS_ON]-(s:Service)
WITH b,count(distinct s) AS c
WHERE c > 1 RETURN b.name,c
```

Figure 4.6: Asynchronous Messaging Query

4.2.4 API Gateway Pattern

The API Gateway [19] or Gateway Routing [20] pattern creates a single entry point for all microservices. This decouples clients from individual microservices, allowing for differences in a public API (the gateway) and a private API (the individual microservice APIs). This is especially useful for versioning. The disadvantage is the complexity of an additional layer.

This pattern can be identified by a single root-level service which depends on more than one other service as in Figure 4.7. The Cypher query can be seen in Figure 4.8.

4.2.5 Backends-for-Frontends Pattern

The Backends-for-Frontends patten is similar to the aforementioned API Gateway, it is a gateway layer providing access to microservices. The difference is there are multiple gateways, which are channel-specific [21]. For example there may be a backend for web clients

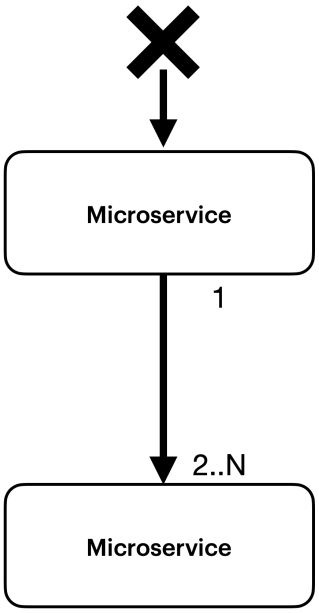


Figure 4.7: API Gateway Pattern Relationship
The X indicates the absence of a parent.

```
MATCH (p:Service)-[:DEPENDS_ON]->(c:Service)
WITH p,COUNT(DISTINCT c) AS count
WHERE NOT (:Service)-[:DEPENDS_ON]->(p) AND count > 1
RETURN p.name,count
```

Figure 4.8: API Gateway Query

and a separate backend for mobile clients. The mobile backend may offer a subset of the data compared to the web client to conserve bandwidth. The advantage is to offer more appropriate APIs for different clients, at the cost of additional complexity.

The pattern can be identified by multiple root-level services which depend on more than one other service as in Figure 4.9. The Cypher query is the same as in Figure 4.8 however we expect to see multiple gateways returned as opposed to only one.

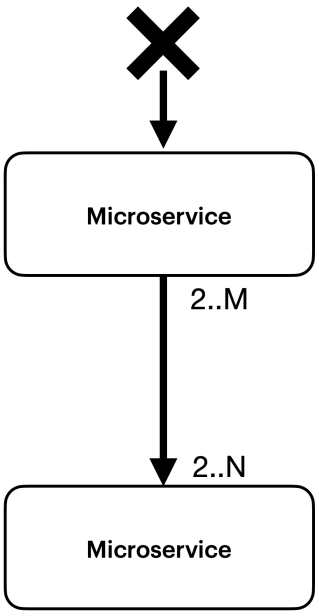


Figure 4.9: Backends-for-Frontends Pattern Relationship

Chapter 5

Evaluation

5.1 Benchmark Systems

To evaluate an architecture recovery technique it is desirable to use a benchmark system for replicability. Nine different microservice benchmark systems were investigated for this purpose and their properties are listed in Table 5.1.

System	Services	Languages	Container	Orchestration
Acme Air	5	Java, Node.JS	None	None
Train Ticket	36	Java, Python, Node.JS, Go	Docker	Kubernetes
Music Store	6	C#	Docker	None
Spring Cloud Demo Apps	6	Java	Docker	None
Bifrost Microservices	5	Node.JS	Docker	None
Socks Shop	8	Java, Go, Node.JS	Docker	Kubernetes
Staffjoy	9	Go	Docker	Kubernetes
eShopOnContainers	8	C#	Docker	Kubernetes
Teacher Management	4	Java, React	Docker	None

Table 5.1: Benchmark Systems

5.1.1 TrainTicket

The TrainTicket system is a railway ticket application presented in 2018 developed at Fudan University [22] and was last updated in May of 2020. TrainTicket aims to address the need for more complex microservice benchmark systems. It has over 60 KLOC of polyglot code

and is decomposed into over 30 microservices. It is also well documented, providing a Wiki describing each microservice, a Swagger UI documenting each microservice's endpoints, and a prescriptive architectural diagram as shown in Figure 5.1. At this time, it is the most complex microservice benchmark system available.

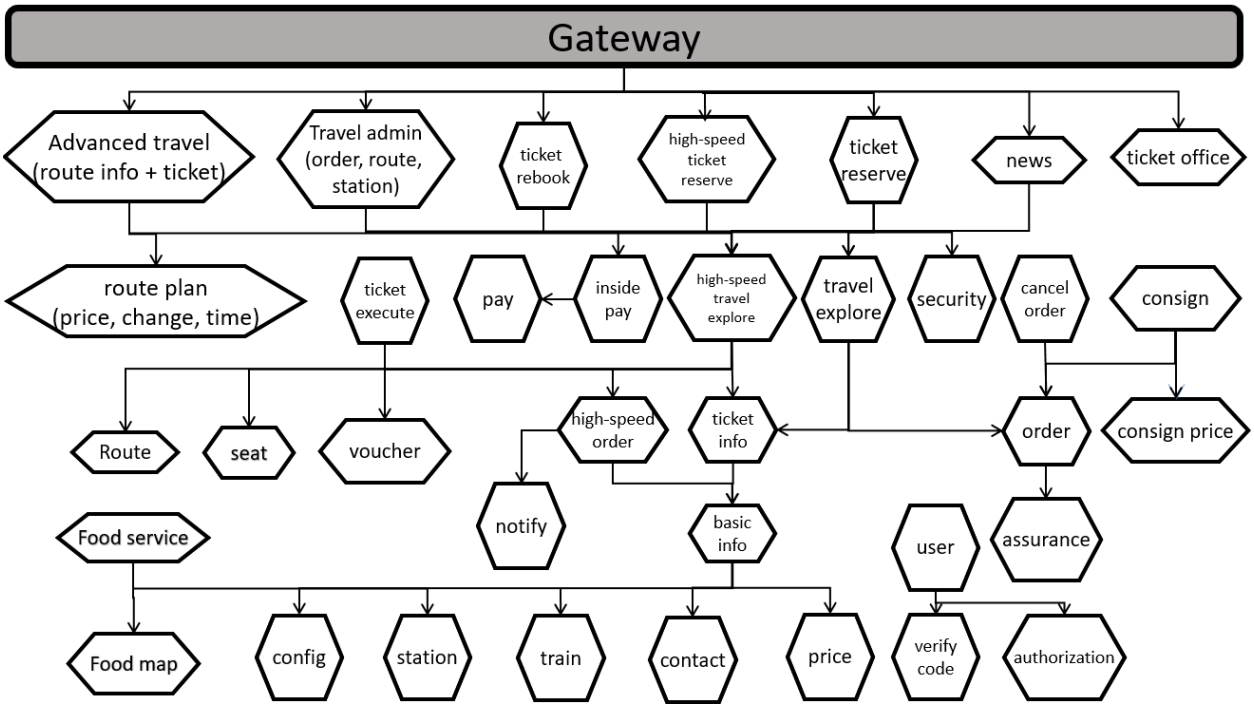


Figure 5.1: Prescriptive Architecture of TrainTicket

TrainTicket Use Cases

Use cases were deduced from TrainTicket's documentation. The project user guide offers a basic workflow of use cases which includes: logging in, searching for train tickets, booking a ticket, paying for the ticket, checking in at the station, and ticket collection. These use cases were scripted as Python code using the `requests` library. To find additional use cases, the project Wiki documentation [23] for each microservice API endpoints was reviewed. First, use cases were scripted for any microservice that was not already hit by existing use cases. Second, use cases were scripted for API endpoints that were not already hit by existing use cases. In the time available, 25 use cases were scripted and they are listed in Table 5.2:

Table 5.2: TrainTicket Scripted Use Cases

Generate CAPTCHA verification code	User login
Get assurance types	Get Food choices
Simple Search	Advanced search
Get contacts for customer	Get Region List
Get News	Rebook ticket
Get Order List	Get High-Speed Order List
Book Ticket	Book High-Speed Ticket
Get Voucher	Pay for ticket
Check in at Station	Collect Ticket
Cancel Ticket	Consign
Admin Get Orders	Admin Get Routes
Admin Get Travel	Admin Get Users
Admin Get Contacts	

5.1.2 Socks Shop

Socks Shop [24] is a microservice demo application created by WeaveWorks. It was last updated on July 17th, 2018. It is unique from the other benchmark systems in that each microservice has its own repository. The prescriptive architecture can be seen in Figure 5.2.

Socks Shop Use Cases

Socks Shop provides a suite of comprehensive load tests which were executed to stimulate network traffic amongst the services.

5.1.3 eShopOnContainers

eShopOnContainers is the reference microservice application from a Microsoft book on architecting containerized .NET microservices [25]. It relies on several Microsoft technologies such as C# and SQLServer but also leverages open-source tools like Redis and RabbitMQ. eShopContainers has over 25KLOC across 8 microservices. It is well-documented, with rea-

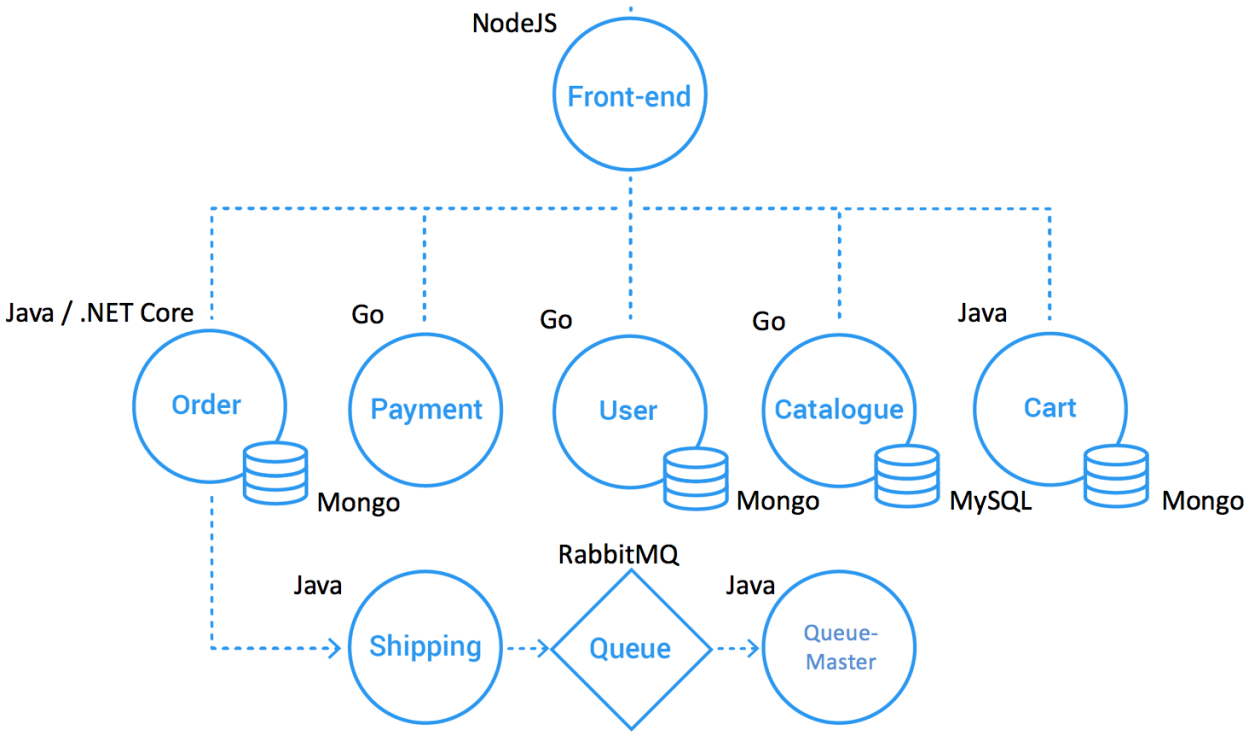


Figure 5.2: Prescriptive Architecture of Socks Shop

soning behind some design choices in the book, and Swagger UI documentation of APIs. Updates on the repository are frequent, the last of which was on May 5th 2021.

eShopOnContainers Use Cases

Use cases were derived from the documentation wiki [26] and thru exploratory testing. Some remaining use cases were found using the Swagger UI documentation for service APIs. The ultimate set of use cases are in Table 5.3.

Table 5.3: eShopOnContainers Scripted Use Cases

Get WebMVC landing page	Get Mobile Landing page
Get Configuration	Get Mobile Catalog
Get Brands	Get Catalog Types
Get Items	Get User Info
Add Item to Basket	Get Basket
Get Order Detail	Get Orders
Wait for Order Completion	

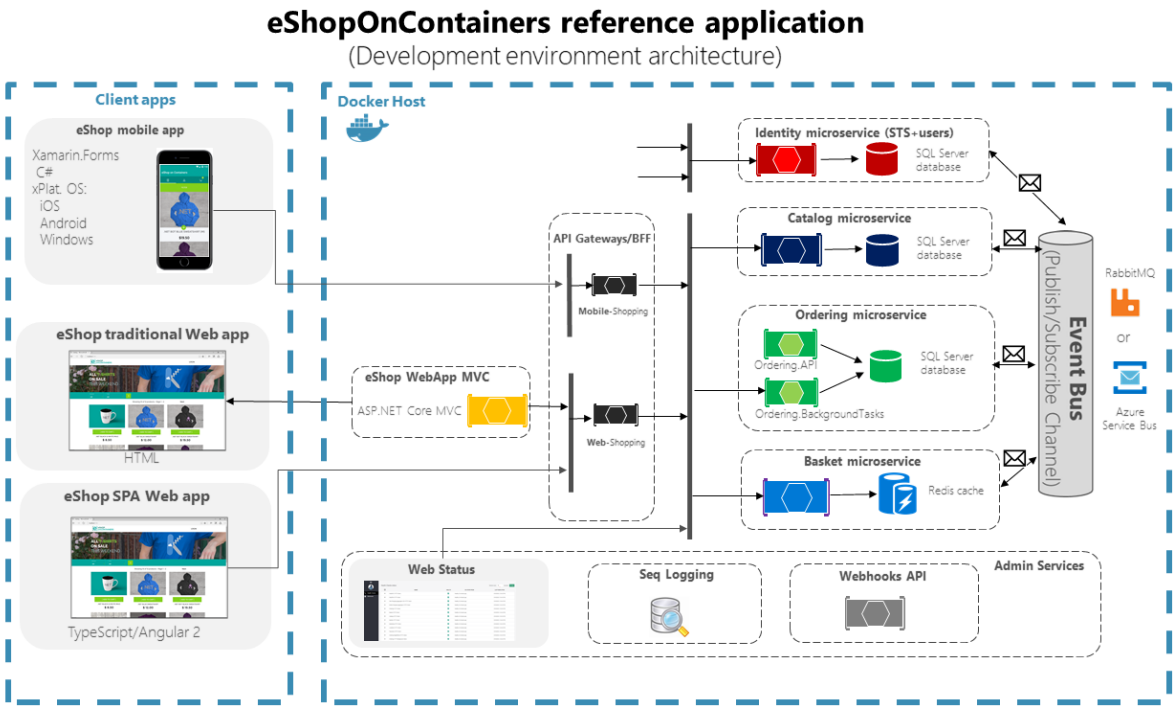


Figure 5.3: Prescriptive Architecture of eShopOnContainers

5.2 Recovery Results

5.2.1 TrainTicket

The resulting GraphViz graph used the K8s service names such as `ts-user-service`. These were renamed to use the same service names in the prescriptive architecture (Figure 5.4). As well, the admin microservices were consolidated to match the prescriptive architecture, this is discussed in the Conformance section below. The resulting recovered architecture can be seen in Figure 5.4.

Since the prescriptive architecture omits utility components, e.g., databases, these were filtered from the recovered architecture also. TrainTicket uses two types of databases, MongoDB and MySQL, so services which contain only `mongo` or `mysql` containers are removed from the analysis.

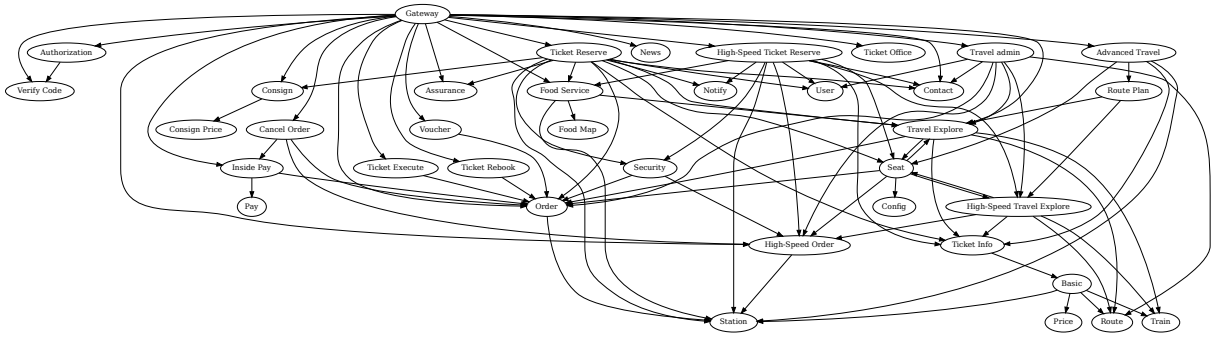


Figure 5.4: Recovered Architecture of TrainTicket

A comparison of the number of components (microservices) and connectors (client-server dependencies) between the prescriptive architecture and the recovered architecture can be seen in Figure 5.4.

	Prescriptive	Recovered	Difference	Same	Missing	Added
Components	37	41	4 (+11%)	37	0	4
Connectors	65	92	27 (+42%)	33	32	61

Table 5.4: TrainTicket Components & Connectors

Figure 5.5 superimposes the adjacency matrix of the prescriptive and recovered architectures. The green elements represent dependencies which are the same in both architectures. The blue elements represent dependencies which were added to the recovered architecture but not in the prescriptive architecture. The red elements represent dependencies which exist in the prescriptive but are missing from the recovered architecture.

There were 33 dependencies which were the same in both architectures (green), 32 dependencies missing (red), and 61 dependencies which were added (blue).

Conformance

Table 5.4 shows that four additional components were recovered that were not represented in Figure 5.1. The reason for this is the *Travel admin* microservice is in fact a collection of

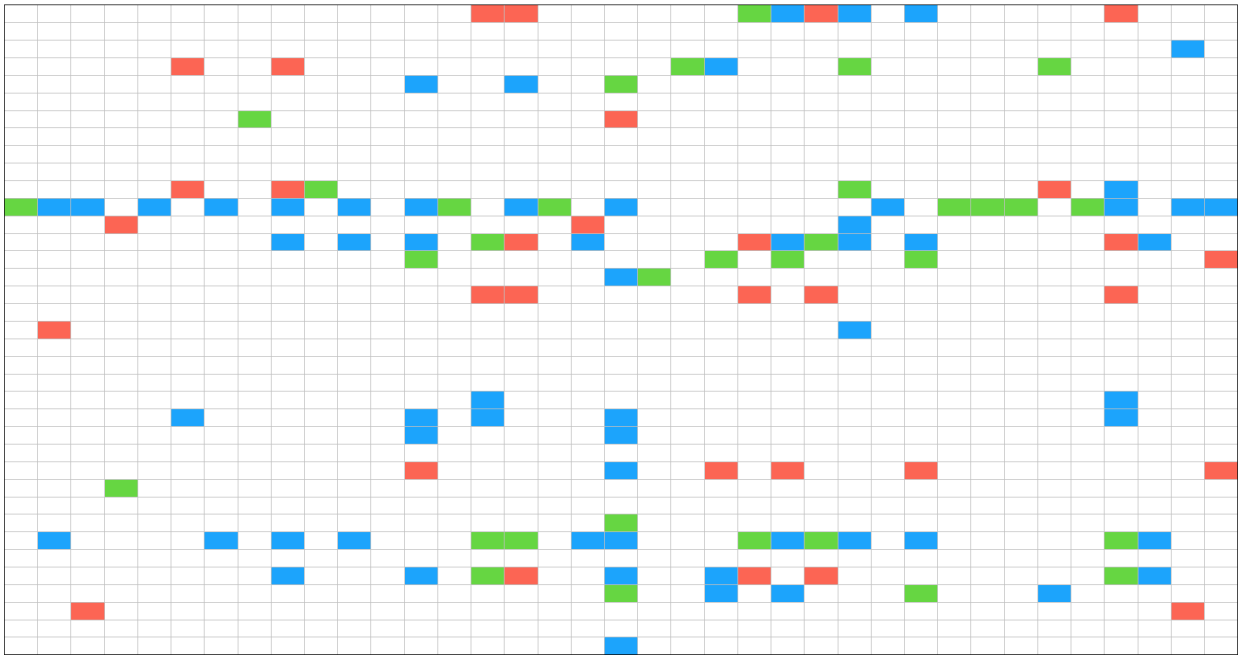


Figure 5.5: TrainTicket Adjacency Matrix Differences.
Red: Missing, Green: Same, Blue: Added.

five microservices: `ts-admin-order-service`, `ts-admin-basic-info-service`, `ts-admin-travel-service`, `ts-admin-user-service`, and `ts-admin-route-service`. The TrainTicket authors may have omitted these services from their architecture graph for simplicity. As mentioned in earlier, these components were consolidated in the recovered architecture to match. This allows for a 1:1 comparison of the connectors.

Table 5.4 shows that there were 27 more dependencies in the recovered architecture. Figrue 5.5 further shows there are a number of connectors both missing from and added to the recovered architecture. These differences are not as concise to explain than the component differences, however some examples are investigated below.

Observe the *News* microservice as seen in Figure 5.1 and as seen in Figure 5.4. Both components are dependencies of the *Gateway*; however, in the prescriptive architecture, *News* has several dependencies, whereas the recovered component has none. When reviewing the API documentation for this microservice, the TrainTicket authors indicate that the service

"only have [sic] test content" [23]. Indeed, when the endpoint of this microservice is requested, it only returns a static test response. This microservice appears to be incomplete and, therefore, is missing dependencies due to only being a stub.

It is observed that the prescriptive architecture generally follows a tree topology i.e. there are no cycles in the architecture graph. However, in the recovered architecture, two cycles were found: between the *Travel Explore* and *Seat* components, and between the *High-Speed Travel Explore* and *Seat* components. This represents an interdependency between the aforementioned microservices. Considering the topology of the prescriptive architecture, and that these are the only two cycles among 92 connectors, it is possible that these interdependencies represent an architectural erosion.

5.2.2 Socks Shop

The comparison of components and connectors between the prescriptive and recovered architecture can be seen in Figure 5.5. The recovered architecture can be seen in Figure 5.6. The adjacency matrix can be seen in Figure 5.7. There were 11 depedencies which where the same, 1 missing, and 4 added.

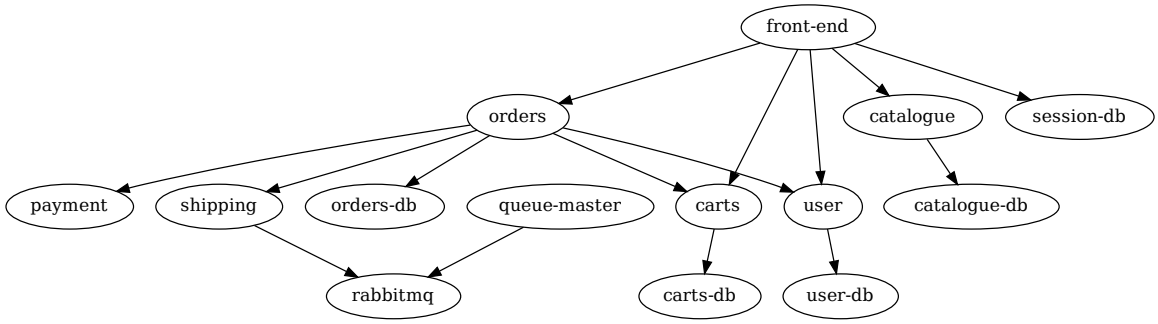


Figure 5.6: Recovered Architecture of Socks Shop

	Prescriptive	Recovered	Difference	Same	Missing	Added
Components	13	14	1 (+8%)	13	0	1
Connectors	12	15	3 (+25%)	11	1	4

Table 5.5: Socks Shop Components & Connectors

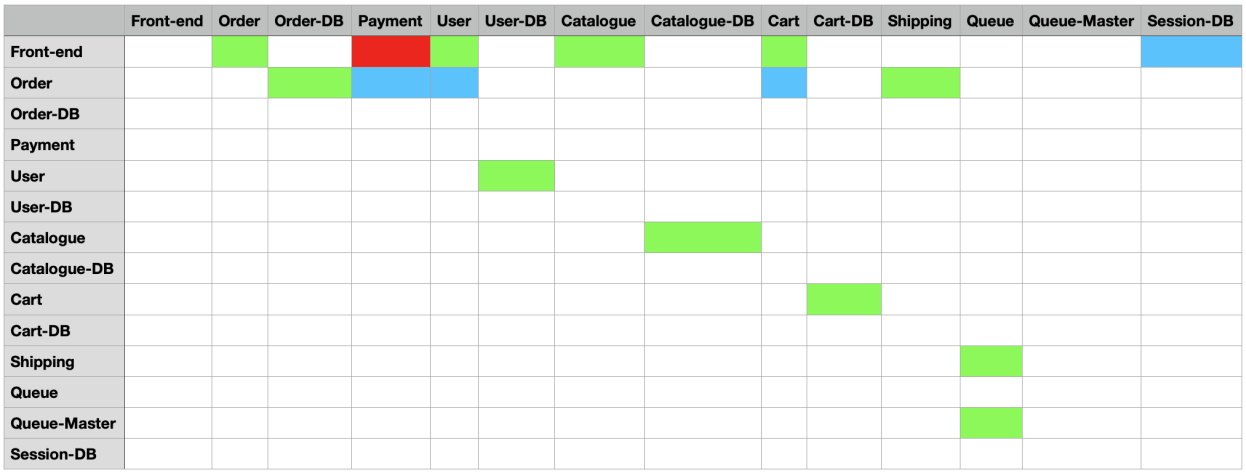


Figure 5.7: Socks Shop Adjacency Matrix Differences.
Red: Missing, Green: Same, Blue: Added.

Conformance

In Table 5.5 there is one additional component compared to the prescriptive architecture in Figure 5.2. This additional component is the `session-db` seen in Figure 5.6. This missing component is likely an oversight in the prescriptive architecture; all the other databases are represented.

The differences in connectors seen in Table 5.5 appear due to the simplified nature of the prescriptive architecture. As seen in Figure 5.6, a dependency hierarchy exists between the `orders` microservice and the `payment`, `cart`, and `user` microservices. These are omitted from the prescriptive architecture. The one missing connector, between the front-end and the payment microservice seen in Figure 5.2, again may be an oversimplification of the architecture.

5.2.3 eShopOnContainers

The prescriptive architecture for eShopOnContainers (Figure 5.3) includes the utility components such as databases and message brokers, so these are left in the recovered architecture in Figure 5.8. The comparison of components and connectors between the prescriptive architecture and the recovered architecture can be seen in Figure 5.6.

An adjacency matrix of the prescriptive and recovered connectors is in Figure 5.9. There were 25 dependencies which were the same in both architectures, 3 dependencies missing, and 14 dependencies which were added.

	Prescriptive	Recovered	Difference	Same	Missing	Added
Components	18	19	(+6%)	17	1	2
Connectors	27	39	(+44%)	25	3	14

Table 5.6: eShopOnContainers Components & Connectors

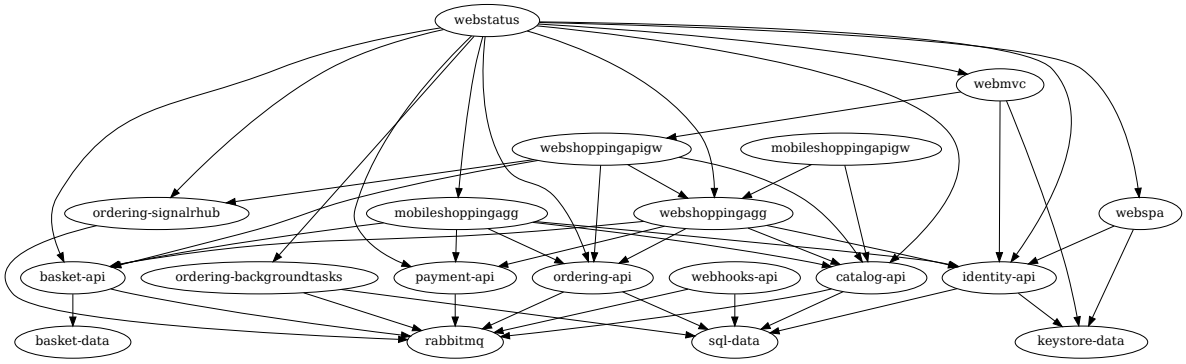


Figure 5.8: Recovered Architecture of eShopOnContainers

Conformance

There were two additional components and one missing in the recovered architecture. The two additional components are the `payment-api` and the `keystore-data`. The `payment-api` is mentioned in the book [25], however, it is omitted from the prescriptive architecture in

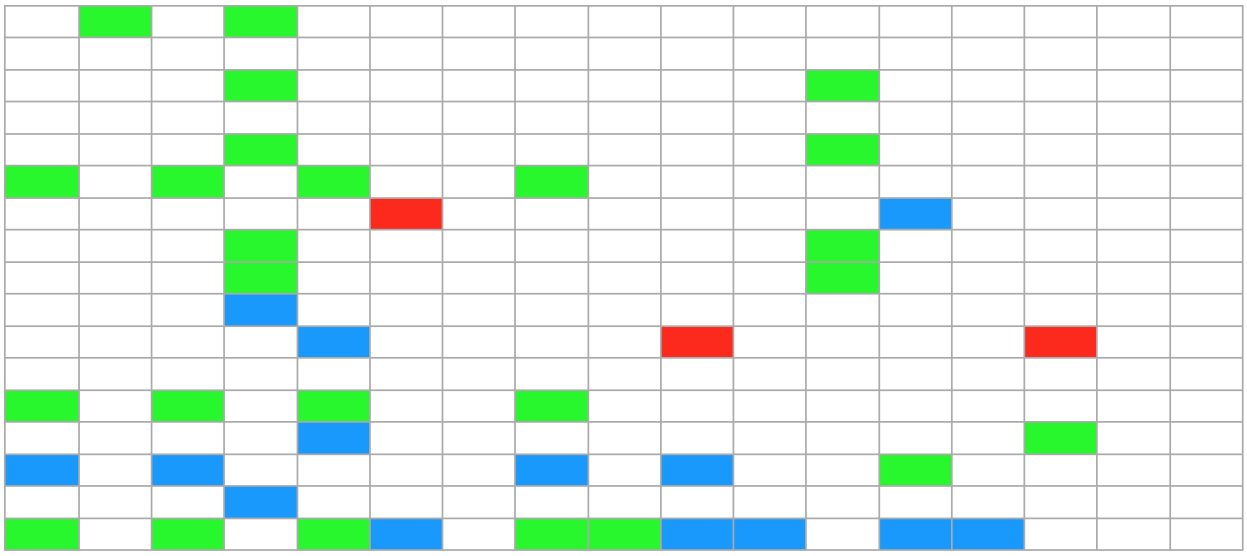


Figure 5.9: eShopOnContainers Adjacency Matrix Differences.
Red: Missing, Green: Same, Blue: Added.

Figure 5.3, this is likely an oversight. `keystore-data` is a Redis cache used by the MVC Web App and SPA Web App, and was perhaps omitted from the prescriptive architecture for simplicity. The missing component is the *Seq Logging* admin service, which is a logging utility that did not appear to be a part of the eShopOnContainers Kubernetes deployment.

As seen in Figure 5.6, there were more connectors recovered than in the prescriptive architecture. For instance, in the recovered architecture, there is a dependency on Identity API (`identity-api`) from the SPA Web App (`webspa`) and the MVC Web App (`webmvc`). The prescriptive architecture in Figure 5.3 shows there are connectors to the Identity API, however, it does not indicate to which components it is connected to. This appears to be an oversimplification in the prescriptive architecture.

A connector missing from the recovered architecture is between the SPA Web App (`webspa`) and the Web-Shopping gateway (`webshoppingapigw`). The Web SPA serves Javascript code that ultimately runs on the user's browser. The user's browser does call the Web-Shopping gateway API but the `webspa` component itself does not communicate with the gateway. Thus, in the prescriptive architecture, this connector represents a data-flow at the client-side

moreso than a server-side dependency, and is why it was not recovered.

5.3 Microservice Pattern Identification Results

The pattern identification queries in Chapter 4 were executed for each benchmark system in Neo4j. Expected instances of patterns were determined from ground truth sources such as documentation, prescriptive architecture, and as well from the recovered architectures. The expected instances versus the actual instances were compared in terms of true positives, false positives, and false negatives. The results can be seen in Table 5.7. The overall precision and recall were computed and can be seen in Table 5.8.

	TrainTicket			**Socks Shop**			**eShopOnContainers**		
Pattern	t_p	f_p	f_n	t_p	f_p	f_n	t_p	f_p	f_n
Shared Database	0	0	0	0	0	0	2	0	0
Database-per	21	0	0	5	0	0	1	0	0
Async. Messaging	0	0	0	1	0	0	1	0	0
API Gateway	1	0	0	1	0	0	0	0	0
Backend4Frontend	0	0	0	0	0	0	1	1	1

Table 5.7: True Positives, False Positives, and False Negatives for each Pattern & System

Pattern	**Precision**	**Recall**
Shared Database	100%	100%
Database-per-Service	100%	100%
Asynchronous Messaging	100%	100%
API Gateway	100%	100%
Backends-for-Frontends	50%	50%

Table 5.8: Overall Precision & Recall Results

5.4 Discussion

Additional components and connectors found in the recovered architectures which were not present in the prescriptive architectures were reviewed and investigated in the respective

source code to look for signs of false positives. No false positives were identified, so the additional recovered components and connectors appear to be accurate. The remaining differences between the prescriptive and recovered architectures observed in Section 5.1 are largely explained by omissions in the prescriptive architecture either for simplicity or possibly oversight. The recovery technique does appear accurate overall, with the caveats given in Section 5.5.

For the pattern identification technique, the results were accurate for 4 of the 5 patterns. The Backends-for-Frontends (BFF) pattern showed only 50% precision. The only benchmark system using this pattern is eShopOnContainers, which has two instances of the BFF pattern: there is a BFF for web clients (`webshoppingapigw`) and a BFF for mobile clients (`mobileshoppingapigw`). The pattern detection correctly identified the mobile BFF, however it missed the web BFF and falsely identified a monitoring service (`webstatus`) as a BFF. `webstatus` was identified since it is a root-level component which depends on multiple services for monitoring their health. If this were a common monitoring tool such as Nagios or Prometheus, the service could have been designated as a utility component, but in the case eShopOnContainers, it was a custom .NET microservice. In the case of the missing web BFF, this is due to the `webmvc` service having a dependency on it, so it was not a root-level service. The `webmvc` component is a legacy-style web UI opposed to a more modern SPA (single-page application) web UI. eShopOnContainers offers both web UI styles for demonstration purposes, but this is uncommon in microservice systems, the other two benchmark systems only use the SPA style UI.

5.5 Threats to Validity

In the cases where manual scripting of use cases was necessary, i.e. in TrainTicket and eShopOn Containers, an internal threat to validity is present: Not all use cases may be

represented. Each microservice is treated as a black box so it is uncertain how much actual code coverage was achieved by the chosen use cases. There may be use cases that would have introduced additional dependencies between the microservices and, if these were not elicited by the chosen use cases, then the recovered architecture may be incomplete.

Additionally, a reliability thread is present. Other researchers attempting to duplicate results for these systems may choose different use cases which elicit different dependencies. While it is reasonable to expect a large overlap, the architectures from this recovery is likely to have some different connectors recovered and thus the architectures will differ.

Lastly, an internal threat exists if the mapping in Table 3.1 is not populated comprehensively.

Chapter 6

Related Work

6.1 Architectural Styles

Taylor et. al in their text define an architectural style, and use a rubric to definite a variety of common architectural styles in the categories of Language-Influenced styles, Layered styles, Dataflow styles, Shared Memory styles, Interpreter syltes, Implicit Invocation styles, and Peer-to-Peer styles [9]. These styles as defined with the authors' rubric were studied to produce the microservices architectural style presented.

6.2 Architecture Recovery

Garcia et al. proposed an architecture recovery framework to obtain ground-truth architectures [27]. Ground-truth architectures are those which have been verified as accurate by the system designers, which is necessary to evaluate the accuracy of recovery techniques. They obtinaed ground-truth architectures for 4 software systems. Garcia et. al used these ground-truth architectures to compare 6 different architecture recovery tools to determine

the strenghs and weakness of different recovery techniques.

6.3 Microservice Architecture Recovery

The recovery technique proposed was in part derived from Granchelli et al. whom performed a dynamic microservice architecture recovery by monitoring network traffic within the system during the execution of manually scripted use cases [5]. Similarly, their technique relied upon the containerization (Dockerization) of the microservices. The technique proposed in this thesis differs in that it relies not only on containerization, but the container orchestration of microservices thru Kubernetes. The advantage of the Granchelli et al. approach is it can be used on systems which do not have a Kubernetes deployment. The disadvantage is that identifying dependencies between microservices requires some manual intervention (other than use case scripting) due to the myriad of possible service discovery mechanisms. Since Kubernetes provides a global service discovery via DNS, we can assume the microservice system to use only DNS and recover the dependencies in a more automated fashion.

Alshuqayra et al. devise a microservice meta-model to aid recovery efforts [6]. In their work, they used both static and dynamic analysis, as well as extracting information from containerization artifacts i.e. Dockerfiles. The static analysis performed was language-dependent, extracting architectural information from Java code. The dynamic analysis was performed using distributed tracing, requiring the code to be instrumented.

6.4 Design Pattern Identification

There is a breadth of research on identifying design patterns in object-oriented systems. Kaczor et. al devised algorithms for identifying design patterns and applied them to three

Java codebases [28]. Guehenuec et al. proposed an improved technique and applied it to nine Java codebases [29].

6.5 Microservice Pattern Identification

Walker et. al devised a technique for identifying architectural smells in microservice systems [8]. For instance, one of the smells they look for is the lack of an API Gateway in larger systems. However, they had no reliable way to detect the API Gateway, instead they give the user a warning if the system maintained over a certain number of services. Ntentos et al. provide foundation for a tool to provide developers feedback during CI of architectural violations in microservice systems, for example identifying cyclic dependencies between components and offering solutions [30]. They evaluated their work not on actual systems but on models. Tighilt et al. proposed a generalized technique for microservice pattern identification relying on containerization and configuration artifacts, but no evaluation is performed [31].

Chapter 7

Conclusion

In this thesis we have presented Microservices as an architectural style. We have proposed a microservice architectural recovery technique which requires fewer manual steps than existing techniques, and applied it to three benchmark systems including the largest benchmark system available, with few innacuracies observed. We have proposed a microservice pattern identification technique and applied it for 5 patterns which occur at an architectural level, four of which were successfully identified.

7.1 Future Work

Improving the detection of the Backends-for-frontends pattern would involve other criteria, such as using not only a dependency subgraph but also the image names as well as the exposed port number, i.e., ports 80 and 443 for HTTP and HTTPS, respectively. There are potentially more microservice patterns which can be detected from an architectural level than the five discussed in this thesis. Exploring these patterns and devising detection criteria for them would be a future research direction.

Bibliography

[1] Liu G., Huang B., Liang Z., Qin M., Zhou H., Li Z., "Microservice: Architecture, Container, and Challenges," IEEE International Conference on Software Quality, Realiability and Security Companion, 2020.

[2] de Santana C.J.L., de Mello Alenar B., Serafirm Prazers C.V., "Reactive Microservices for the Internet of Things A case study in Fog Computing," ACM/SIGAPP Symposium on Applied Computing, April 2019.

[3] Allied Market Research, "Microservices Architecture Market," Feb 2020. [Online]. Available: `https://www.alliedmarketresearch.com/microservices-architecture-market`

[4] Garcia J., Ivkovic I., Medvidovic N., "A Comparative Analysis of Software Architecture Recovery Techniques," IEEE/ACM International Conference on Automated Software Engineering, 2013.

[5] Granchelli. G, Cardarelli M., Di Francesco P., Malavolta I., Iovini L., Di Salle A., "Towards Recovering the Software Architecture of Microservice-based Systems," IEEE International Conference on Software Architecture Workshops, 2017.

[6] Alshuqayran N., Ali N., Evans R., "Towards Micro Service Architecture Recovery: An Empirical Study," IEEE International Conference on Software Architecture, 2018.

[7] Osses, F., Marquez G., Astudillo, .H, "Exploration of Academic and Industrial Evi-

dence about Architectural Tactics and Patterns in Microservices," 40th International Conference on Software Engineering. 2018.

[8] Walker A., Das D., Cerny T., "Automated Code-Smell Detection in Microservies Through Static Analysis: A Case Study," Applied Sciences, Vol. 10 Iss. 21, 2020.

[9] Taylor R., Medvidovic N., Dashofy E., "Software Architecture: Foundations, Theory and Practice," Wiley, 2009.

[10] Lenarduzzi V, Lomio F., Saarimaki N., Taibi D., "Does migrating a monolithic system to microservices decrease the technical debt?", Journal of Systems & Software, 2020.

[11] Villamizar M., Garces O., Castro H., Verano M., Salamanca L., Casallas R., "Evaluating the Monolithic and the Microservice Arhicture Patterns to Deploy Web Applications in the Cloud," Computing Columbian Conference, 2015.

[12] Koschel A., Astrova I., Dotterl J., "Making the Move to Microservice Architecture," International Conference on Information Society, 2017.

[13] Villamizar M., Garces O., Ochoa L., Salamanca L., Verano M., Casallas R., et al., "Infrastructure Cost Comparison of Running Web Applications in the Cloud using AWS Lambda and Monolithic and Microservice Architectures," IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 2016.

[14] Garcia J., Popescu D., Edwards G., Medvidovic N., "Toward a Catalogue of Architectural Bad Smells", International Conference on the Quality of Software Architectures, 2009.

[15] Boettiger C., "An Introduction to Docker for reproducible research," ACM SIGOPS Operating Systems Review, January 2015.

[16] Al Jawarneh I., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," IEEE International Conference on Communications, 2019.

[17] Kronmueller M., Chang D., Hu H., Desoky A., "A Graph Database of Yelp Dataset Challenge," IEEE International Symposium on Signal Processing and Information Technology. 2018.

[18] Neo4j Inc., "What is openCypher?" 2018. [Online]. Available: `https://www.opencypher.org`. Accessed: 05-May-2021.

[19] Richardson, C., "Microservice Architecture," 2020. [Online]. Available: `https://microservices.io`. Accessed: 05-May-2021.

[20] Wasson M., "Design patterns for microservices," 2017. [Online]. Available: `https://azure.microsoft.com/en-us/blog/design-patterns-for-microservices/`. Accessed: 05-May-2021.

[21] Brown K., Woolf B., "Implementation Patterns for Microservice Architectures," Conference on Pattern Languages of Programs. 2016.

[22] Zhou X., Peng X., Xie T., Sun J., Xu C., Ji C., et al., "Benchmarking Microservice Systems for Software Engineering Research," International Conference on Software Engineering Compantion, June 2018.

[23] Fudan Software Engineering Lab, "Train Ticket Wiki," 2020. [Online]. Available: `https://github.com/FudanSELab/train-ticket/wiki`. Accessed: 03-Mar-2021.

[24] WeaveWorks, "Sock Shop: A Microservice Demo Application," 2018. [Online]. Available: `https://github.com/microservices-demo/microservices-demo`. Accessed: 05-May-2021.

[25] de la Torre C., Wagner B., Rousos M., ".NET Microservices: Architecture for Containerized .NET Applications," Microsoft Developer Division, 2020.

[26] .NET Foundation, "Explore the application," 2019. [Online]. Available:

https://github.com/dotnet-architecture/eShopOnContainers/wiki/Explore-the-applicatio
Accessed: 05-May-2021.

[27] Garcia J., Krka I., Mattmann C., Medvidovic N., "Obtaining Ground-Truth Software Architectures," International Conference on Software Engineering. 2013.

[28] Kaczor O., Guehenuec Y., Hamel S., "Identification of design motifs with pattern matching algorithms," Journal of Information and Software Technology. 2009.

[29] Gueheneuc Y., Guyomare'h J., Sahraoui H., "Improving design-pattern identification: a new approach and an exploratory study," Software Quality Journal. 2009.

[30] Ntentos E., Zdun U., Plakidas K., Geiger S., "Semi-automatic Feedback for Improving Architecture Conformance to Microservice Patterns and Practices," 18th IEEE International Conference on Software Architecture. 2021.

[31] Tighilt R., Abdellatif M., Abu Saad N., Moha N., Gueheneuc Y., "Collection and Identification of Microservices Patterns And Antipatterns," Proceedings of the 12th Conference Francophone sur les Architectures Logicielles (CAL). 2019.