UNDERSTANDING THE PATTERNS OF MICROSERVICE INTERCOMMUNICATION FROM A DEVELOPER PERSPECTIVE

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Anas Nadeem

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

November 2022

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

UNDERSTANDING THE PATTERNS OF MICROSERVICE INTERCOMMUNICATION FROM A DEVELOPER PERSPECTIVE

By

Anas Nadeem

The Supervisory Committee certifies that this thesis complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Muhammad Zubair Malik

Chair

Dr. Zahid Anwar

Dr. María de los Ángeles Alfonseca-Cubero

Approved:

17 November 2022	Dr. Simone Ludwig
Date	Department Chair

ABSTRACT

Microservices Architecture is the modern paradigm for designing software. Based on the divide-and-conquer strategy, microservices architecture organizes the application by furnishing it with a fine-level granularity. Each microservice has a well-defined responsibility and multiple microservices communicate with each other toward a common goal. A momentous decision in designing microservices applications is the choice between orchestration or choreography-based modes as the underlying intercommunication pattern. Choreography entails that microservices work autonomously while orchestration entails that a central coordinator directs the interaction between services. We arbitrate this decision from a developer's perspective by empirically evaluating the properties of a benchmark system mapped into both orchestration and choreographed topologies. In this research, we document our experience from implementing and debugging this system. Our studies demonstrate microservices composed using orchestration exhibit desirable inherent characteristics that make microservice code easier to implement, debug, and scale.

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to the following people who made this research and my master's degree possible. First and foremost, I would like to express my appreciation to my parents who have helped me become the person I am today. It would not have been possible without their support. I also express my whole-hearted gratitude to my advisor, Dr. Zubair Malik for his support and encouragement throughout my academics, research, and career. I would like to extend my sincere thanks to all of my family and friends; who have constantly been supporting me despite the distances. Finally, I would like to thank my friend Nimrah for her constant support.

DEDICATION

This thesis is dedicated to my parents.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MSA	Microservice Architecture
HTTP	Hyper Text Transfer Protocol
REST	Representational State Transfer
SOA	Service-Oriented Architecture
OOD	Object Oriented Design
IP	Internet Protocol
AI	Artificial Intelligence
API	Application Programming Interface
JVM	Java Virtual Machine
gRPC	gRPC Remote Procedure Calls (recursive)
UI	User Interface
DSL	Domain Specific Language
GUI	Graphical User Interface
XML	Extensible Markup Language
SQL	Structured Query Language
CI	Continuous Integration
CD	Continuous Deployment
NoSQL	Non-Structured Query Language
IoT	Internet of Things
TD	Technical Debt

1. INTRODUCTION

1.1. Overview

Microservice Architecture (MSA) is a method of structuring an application by breaking it down into small, cohesive units that are loosely coupled and functionally independent. Their lightweight nature allows the development, deployment, testing, and scaling of these microservices independent from each other [56] making them promising for cloud-native infrastructure. As a result, large web companies including Netflix, Amazon, IBM, and Microsoft have made efforts to evolve their applications towards a microservice-based architecture [14, 16]. The microservice architecture inherits concepts and principles from the Service-Oriented Architecture; both decompose the systems into services available over a network [11], however, the microservice architecture takes it a level up as each microservice may communicate over an entirely different protocol and could be written in entirely different languages.

Adopting microservice architecture is not a straightforward task [7, 21, 44, 53] and requires several critical decisions. One contentious architectural decision is whether to compose the microservices using choreography or orchestration patterns. This decision has vast impacts on the lifecycle of a system and proposes further challenges in its entirety. In this work, we try to develop a strong understanding of these patterns and their challenges from the perspective of a developer. This research is an extension of our previous research [32].

1.2. Motivation

The choice between composing a microservice-based system using either choreography or orchestration is critical. Regardless of the underlying architecture, or collaboration strategy the user needs to be abstracted of the inherent complexities and should be able to receive a seamless experience. This naturally shifts the burden of managing the associated complexities onto the developers [26], who are responsible for building, maintaining, and troubleshooting these systems. This motivates us to explore the pros and cons of both approaches from a developers' perspective. This is a gap in the existing literature [28, 53] as existing research considers only the inherent properties of the system itself, and not the developers' aspect [28, 35].

1.3. Thesis Statement

Systems composed using microservice orchestration are more favourable for a developer in terms of availability of tools, better quality of code and is simple to debug.

In this section we describe our research questions, highlight our contributions and our objectives for carrying out this research.

1.3.1. Research Questions

The goal of the research is to study and highlight the inherent characteristics of the choreography and orchestration approach. In specific, we want to find out how both of these approaches impact the overall developer's experience from implementation, debugging and maintenance. The following are our research questions:-

- **RQ1:** How does the choice of the interaction pattern affect the implementation process of the systems?
- **RQ2:** What are the immediate pros and cons of selecting either of the approaches?
- **RQ3:** How does the choice affect the process of debugging the system?

1.3.2. Research Contributions

To answer the research questions above, We make the following research contributions.

- We dive deep into the literature to understand the problem with microservice choreography and orchestration.
- We discuss our experience in implementing microservice orchestration and choreography techniques on a benchmark microservice system.
- We inject faults into the benchmark system and conduct an empirical study and compare the times required to debug these faults in both orchestration and choreography schemes.

1.3.3. Research Objectives

We conduct a case study around a benchmark system [58], designed for activities around booking train tickets. We consider this benchmark system mapped in choreography as well as orchestration patterns. Our objectives are highlighted as follows:-

- Understand and identify the challenges of the microservice architecture
- Understand the concepts and differences between the microservice orchestration and choreography patterns.
- Consider factors including learnability, and ease of use and present our findings in an organized approach.
- Use scientific methods to gather substantial evidence around both techniques and see how different processes would be carried out in both approaches.
- Based on our results, draw conclusions supported with evidence.
- Provide sufficient equipment for the purpose of expanding our work.

1.4. Thesis Organization

In Chapter 2 we discuss the background of the microservice architecture and the available patterns of communications. In Chapter 3 we document some concepts and clarify some definitions, disambiguate between any concepts or jargon and discuss the state of the art in microservices, that were employed while carrying out the experimentation. Further, we highlight the details of our experimentation in Chapter 4 and discuss the results in Chapter 5. Finally, Chapter 6 concludes our research and the findings.

2. LITERATURE REVIEW

2.1. Background

To develop systems, various architectures have been predominantly used in the past such as Data-centric architecture, and Peer-to-Peer architecture. We discuss the most common of these architectures; the Monolithic Architecture and highlight its challenges to develop an understanding of the need for microservice architecture.

2.1.1. Traditional Architecture

Computer applications have evolved from the traditional monolithic architecture in which the entire application [14], including the user interface and the business implementation were housed on the same machine. From a developer's perspective, tasks such as implementing, testing, and debugging such environments were easy as there was better traceability of requests because the components resided on the same machine.

2.1.2. Challenges of the Traditional Application Architecture

Towards the downside, the monolithic application would soon grow into a gigantic system that was difficult to maintain. This also meant that the modules could not be executed, or deployed independently [15]. As a result, the system components were intertwined in a dependency-hell [29, 50] making the application prone to a single point of failure where one component failing could cause the entire system to fail. Despite its challenges, the monolithic architecture was widely used in the past by companies like Amazon and Ebay [14] and is still commonly used for small-scale applications. However, systems requiring high scalability and availability have shifted onto the more promising, microservice architecture [36]. Figure 2.1 demonstrates application components arranged in a monolithic style.

2.1.3. Microservice Architecture

The modern Microservice Architecture paradigm solves the challenges of monolithic architecture. It is an approach to developing the system as a suite of small independent services,

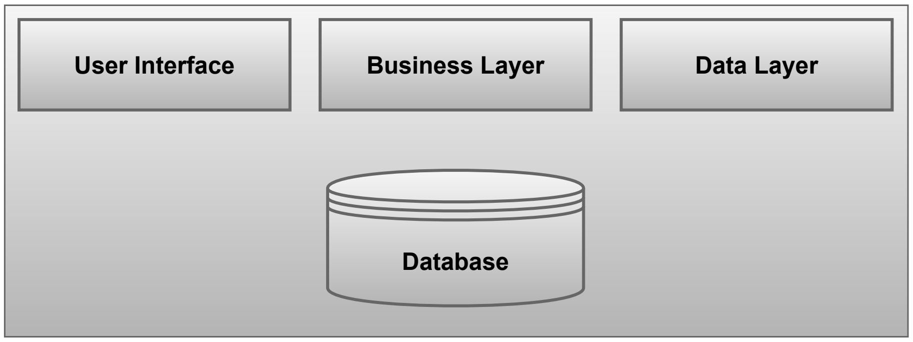


Figure 2.1. Layout of a monolithic application having all components housed in a single unit.

demonstrated in Figure 2.2. These services run in their own independent processes [11] and interact with each other to perform business processes. This makes the system fault-tolerant as a failing microservice can be easily replaced and the other parts of the system can function normally even while that service is down [38]. The microservice architecture solves the challenges of monolithic architecture by introducing the principles of high cohesion, less collision, and scalability.

2.1.4. Challenges of Microservice Architecture

The benefits of microservices are no free lunch [54] and microservices come with several challenges of their own. Microservice-based applications soon become very complex as a result of high dynamism, failing services, a high amount of parallelism, and developers' lack of observability and pose unique challenges [55]. Debugging a microservice system is degrees of magnitude more complex than a traditional system as logs are distributed among multiple isolated services. Microservices typically run inside Docker containers [29], which are lightweight runtimes. Adding an automatic layer of scalability, these container instances are dynamically created or destroyed by container orchestrators such as Kubernetes [9] based on traffic or other metrics. This further complicates the problem as it becomes difficult to map requests to a physical instance of a microservice [56]. Therefore, it is important to make correct architectural choices while composing microservices as adopting anti-patterns and wrong patterns may have adverse effects on the performance, scalability, and cost of the systems [40].

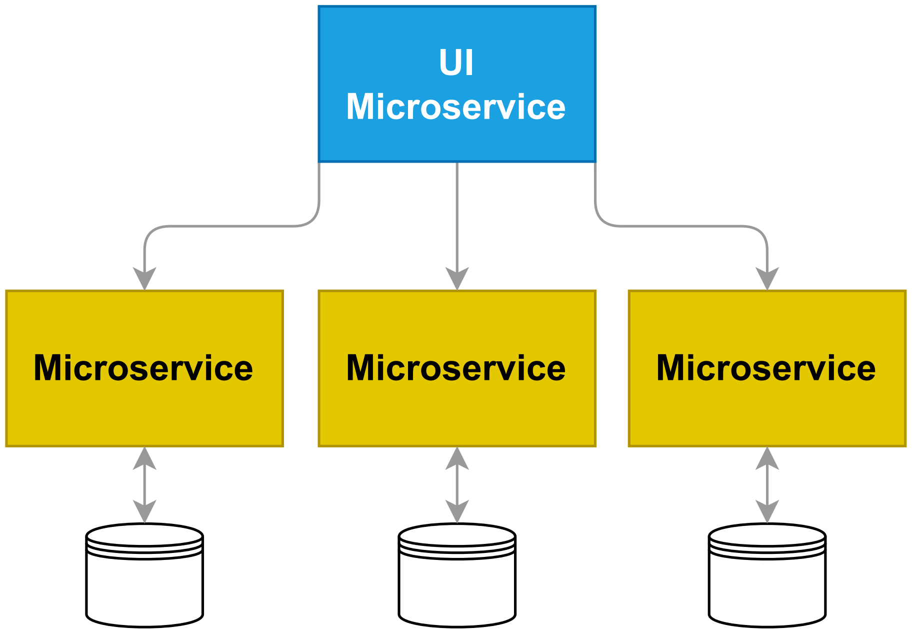


Figure 2.2. Layout of a microservice application where each microservice has its database and communicates with each other.

2.1.5. Composition Styles

A complex microservice-based system might be composed of thousands of microservices [55]. As discussed earlier, each microservice in the system has a well-defined responsibility and performs specialized tasks. Hence, a business request invocation might span across multiple microservices [39]. The collaboration of microservice towards a goal is often termed as a 'workflow'. This invocation or collaboration in a microservice system is composed using either microservice choreography or microservice orchestration [5, 28, 32, 35, 39, 42]. The selection of the right composition strategy is a key decision while architecting microservice systems which impacts developer activities such as debugging, testing, and trouble-shooting.

2.1.6. Choreography

Microservice choreography is traditionally the most prevalent approach while composing microservices [19, 39]. Each microservice operates autonomously hereby following the event-driven paradigm. Hence, the flow of a request is from one microservice to another until a response is generated, which is communicated back to the requester [28]. Figure 2.3

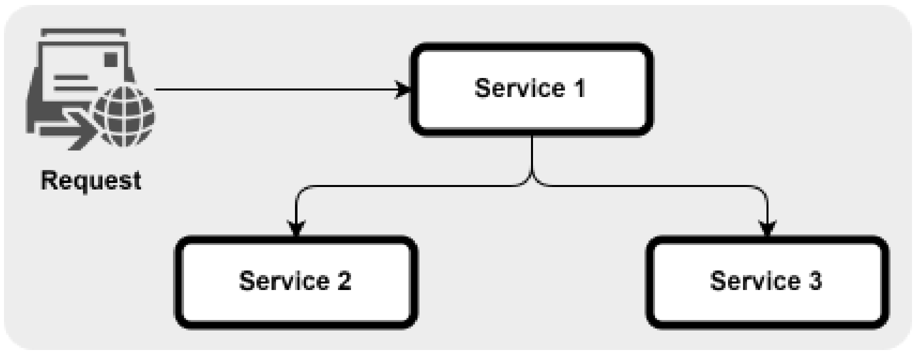


Figure 2.3. Request flow in a microservice system composed using choreography.

represents the underlying topology of this communication. Commonly, in choreography, each microservice listens and responds to events from a message broker such as Apache Kafka [18], ActiveMQ [43], etc. In this work, we mainly consider the topological aspect of this communication, and rather than using complex distributed queues, we choreograph our microservices using REST, which is another common way of organizing the choreography pattern [13]. Choreography has no hard dependencies and each microservice is independent in handling business, including enabling the service with monitoring, fault tolerance, and fault resilience capabilities.

2.1.7. Orchestration

Microservice orchestration uses a central coordinator that acts as the brain of communication within a microservice-based system. Each request and response is directed to and from the orchestrator [28, 39]. Figure 2.4 demonstrates the flow of a request within a microservice orchestration scheme. The property of control flow being centrally controlled, and the microservices not being atomic allows the construction of frameworks to enable the orchestration approach and allowing to handle monitoring, and fault tolerance directly from the orchestrators.

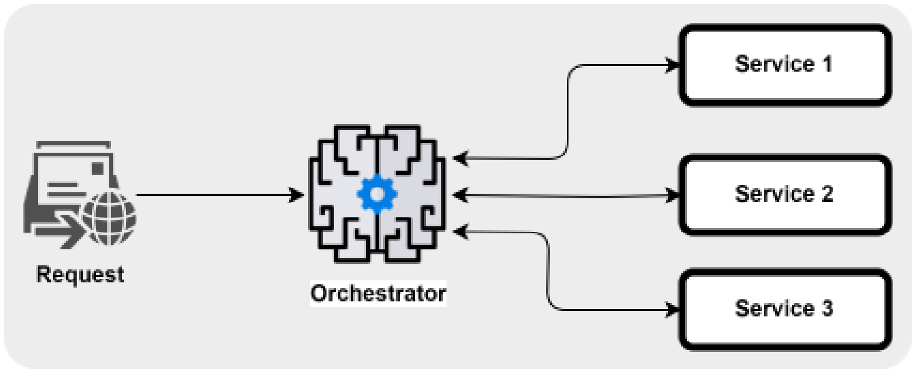


Figure 2.4. Request flow in a microservice system composed using orchestration.

2.2. Related Work

2.2.1. On Microservice Architecture

Microservice architecture, which was designed to meet the shortcomings of the monolithic architecture [20] has received a lot of attention over the last decade. Developers are making effort to migrate their monolithic applications onto the microservice framework thus enabling the applications to become cloud native [4]. This migrating trend towards the cloud and its increasing adoption has made it common for deployments to rely on container-based microservices [41]. Studies around microservices [41] have proved them to be promising for companies allowing to manage large code bases. Moreover, the microservice architecture has not only evolved the method of engineering but has impacted the agility of an organization where the teams solely manage one or multiple microservices [22], hence allowing efficient division of labor.

Microservices, due to their lightweight nature, have seen their adaptation in the Internet of Things (IoT) [10] and edge computing [52] with critical use cases. Their efficiency has resulted in their utilization not only in the software industry but they have also helped scale systems in the area of biotechnology [8]. Monolithic applications are increasingly being broken down into microservices. According to a study, this transition is worthwhile as microservice-based systems reduce infrastructure costs by at least 70% [51] when compared with monolithic applications. Makris et. al [26] highlight the challenges of this transition. More recent work is around devising strategies to formalize and automate the process of this transition [1, 46].

2.2.2. On Maintenance and Debugging of Microservice Based Systems

The puzzling complexity of system visibility of microservice systems has provoked researchers to ponder ways around solving this challenge. Distributed tracing [34] is a way of having a bird's eye view of the request life-cycle from its origin to its completion. Zhou et. al [55, 57] worked on solving the problem by using the delta-debugging algorithm to devise a strategy for identifying the root causes of failures. Gan et. al [17] worked on the performance debugging aspect of these systems. Zhou et. al [56] also worked on collecting insights from a detailed industrial survey. They presented common faults found in microservice systems and injected these faults into a benchmark system. They further presented results from debugging these faults and the underlying tools used to debug the faults. In our work, we utilize the same benchmark system while conducting our empirical study.

2.2.3. On Orchestration and Choreography

While microservice orchestration and choreography are extensively used in the industry for the composition of microservices, they have received little attention in academia. Although several researchers have tried to arbitrate the decision between making this decision, they did not consider the developer aspect of it [28]. Alan et. al proposed a formal survey methodology for narrowing down the decision of choosing the desired composition strategy. They assign weights to the desired properties and score specific characteristics when mapped to either approach and come to a conclusion based on the maximum points earned based on the weights. C. K. Rudrabhatla [39] worked on a comparison of both patterns. Their findings suggested that microservice choreography is the desired approach for systems involving a small number of microservices however, as soon as the size of the system tends to grow, choreography becomes hard to code, and maintain. More recent work is around combining the best of both approaches to be able to leverage the benefits of both in a hybrid approach [49]

3. RELEVANT TOOLS AND DEFINITIONS

3.1. Tools and their Utilization

We use this section to define the tools that we used in the research and discuss their specific use in detail during our experimentation.

3.1.1. Docker

Docker Container is a lightweight virtualization platform that the industry uses to share resources without worrying about dependencies [2]. This lightweight makes them suitable to be hosted on cloud infrastructure. In our work, we use microservices hosted in Docker containers that run within Docker runtime.

3.1.2. Spring Boot Framework

Springboot framework, is a set of java libraries that facilitate the creation of stand alone microservices [37] by embedding REST capability, database integration and interface with Docker directly. The majority of the microservices we use are written using this framework.

3.1.3. Workflow Engines

Workflow engines are tools, designed to orchestrate microservices. These tools relieve the developer of managing low level tasks such as managing the distributed architecture, handling faults, handling communication and focus on implementing the business logic. These workflow engines include Netflix Conductor [33], Uber Cadence [48]. We use one such workflow engine, Temporal [27], to facilitate orchestration of microservices during our empirical study.

3.2. Definitions

3.2.1. Container Orchestration

Container Orchestration; not to be confused with microservice orchestration, is a technology that automatically up-scales or down-scales container instances. This up-scaling and down-scaling is based on a set of tools defined in that orchestrator by an administrator, usually based on traffic, for load-balancing purposes. These tools are such as Docker Swarm [45], Kubernetes [9].

3.2.2. Fault Tolerance

Fault tolerance refers to the system's ability to handle any fault and recover from it. High availability is an important aspect in distributed systems involving microservices therefore requires immediate attention [23]. We will further discuss this aspect and its manageability for both orchestration and choreography patterns.

3.2.3. Visual Tracing

Visual tracing or trace analysis refers to ways of visualizing the life cycle of a request in a distributed system, typically with the help of graphs. We use the visual tracing technique while debugging our system during our study.

4. APPROACH OF THE STUDY

Due to their complexity, microservice systems are hard to implement and debug, therefore it is important to study how the choice of either choreography or orchestration affects the overall developer experience. While several external factors impact the overall developer experience including team communication, and organizational changes [12], we limit the scope of our study to intrinsic properties of the system itself, including the availability of tools and technical debt of the code produced [12].

Technical debt is a critical issue in the modern world software development industry and can lead to high software quality issues. Technical debt refers to the potential long-term effect of immature code produced during the life-cycle and the 'debt' has to be repaid later in the development process [6], *significantly impacting the developer experience*. Several factors that procure technical debt include application-level code smells such as duplicated code, and tight coupling of components.

First, we report the implementation aspect of this experience including the technical debt of the produced code, and the infrastructure tools and support [12]. Then we discuss the debugging aspect of this system and finally compare the system with both approaches.

4.1. Benchmark System

For the purpose of our study, we use a microservice benchmark system TrainTicket [56]. This benchmark system allows a user to perform typical train ticket booking activities including purchasing, canceling, and changing tickets. TrainTicket, to the best of our knowledge, is the largest publicly available benchmark microservice system which contains over 30 fine-grained microservices. These microservices are written in several languages including Java, NodeJS, Go, and Python. The benchmark system extensively utilizes the Springboot framework, which serves to make the implementation of microservices easier. Additionally, the benchmark system contains common fault cases that occur in industrial microservice systems, gathered from an external industrial survey.

4.1.1. System Configuration

The benchmark system is available at our TrainTicket-workflows GitHub repository [30]. The microservices were hosted on Docker containers running on a Linux machine and were powered by a processor configuration of 3.60 GHz x 8 cores and 32GiB available memory.

4.2. Infrastructure and Tools

4.2.1. Choreography

The benchmark system code is originally composed using the choreography topology. Hence a request in the original TrainTicket system originates from one microservice and traverses from one microservice to another until a response is received. In TrainTicket, microservice choreography is implemented using direct REST API communication. During the case study, mapping TrainTicket to use an event-driven broker such as Apache Kafka was also considered, however, it added little value since it would've only complicated the communication with little to no benefit since the microservice topology would have remained the same.

We observed a low availability of available tools for choreography as compared to orchestration. We document the possible reasons for it in the discussion section.

4.2.2. Orchestration

When mapping to orchestration, we were able to leverage several existing tools. For the purpose of writing the business processes, we employed a workflow engine. These workflow engines relieved us of low-level distributed system management tasks such as handling failures and enabled us to implement only the business logic. After considering variety of workflow engines including Netflix Conductor [33], Uber Cadence [48], and Apache Airflow [3], we opted Temporal [27] due to the following reasons:-

1. We were able to define the microservice communication in plain code rather than using any complex Domain-Specific Language.

2. Temporal itself came with off-the-shelf tool support to ease debugging.

Below we highlight our experience in using Temporal to orchestrate our microservice in detail.

4.2.2.1. Temporal

Temporal workflow engine manages the workflow lifecycle and provides a fault-oblivious stateful programming model to orchestrate microservices. Using Temporal, programmers can write their microservice communication workflows in a familiar language such as Go, Java, and PHP and the platform takes responsibility for managing tasks including fault tolerance, and frees the developer to focus on actual business logic implementation rather than spending time on re-inventing the wheel.

4.2.2.2. Transitioning to Temporal

While mapping code to Temporal, the code had to undergo several changes such as separating business logic from control flow. While the platform promised low-level handling of distributed tasks, the code had to go through several refactoring. The framework had two main constructs to carry out business processes; workflows and activities.

In Temporal, every business process must instantiate a workflow. This workflow manages all the interactions of microservices. Every microservice interaction must be managed by the Temporal framework.

Every non-deterministic action that is prone to failure, such as calls to microservices and calls to external APIs must be wrapped in an activity. These activities are used within the workflow. The state of every input and output from these activities is stateful and is recovered in case of failures enabling fault resilience.

These workflows and activities are then registered using a worker which is responsible for fetching the tasks from the queue and thus executing them.

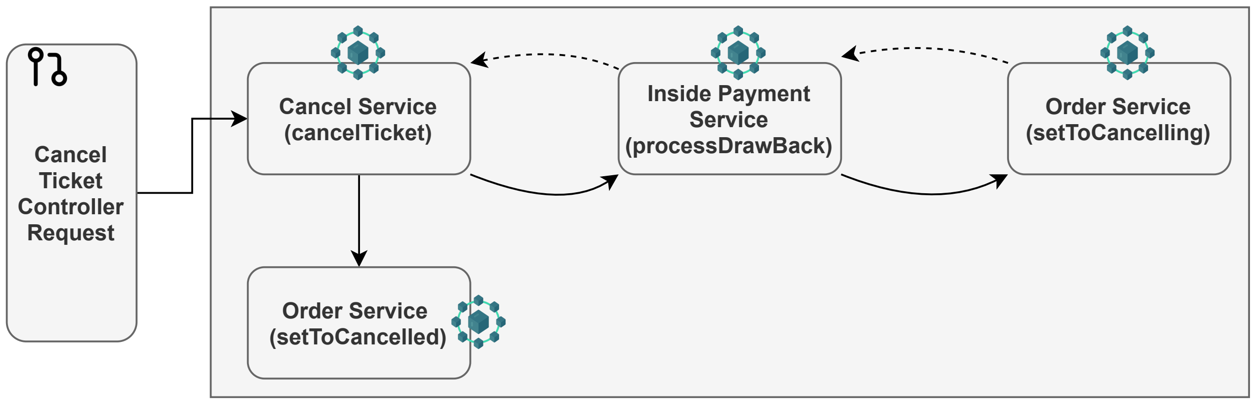

Cancel Ticket Controller Request
Cancel Service (cancelTicket)
Inside Payment Service (processDrawBack)
Order Service (setToCancelling)
Order Service (setToCancelled)

Figure 4.1. Original flow of the business process before mapping to Temporal.

Below we highlight our experience from mapping a microservice business from TrainTicket to Temporal orchestrated workflow, and use a similar strategy to map all the workflows to Temporal.

4.2.2.3. Original Ticket Cancellation Process

The ticket cancellation process in the benchmark system allows canceling a purchased Ticket. The purchase cancellation involves two main operations. The first is processing the refund and the second involves changing the status of the order. When a user requests the cancellation of a ticket, cancel ticket service is involved which calls inside payment service to process the refund. The inside payment service internally triggers the order service to set the order to cancel. before it starts processing. Once the inside payment service completes the processing of the refund, it transfers the control back to cancel service which again calls the order service to set the order to canceled. This whole process is demonstrated in Figure 4.1.

4.2.2.4. Conversion to Temporal Workflow

We were able to map the above business process to a workflow, systematically. First, we mapped the business process to cancelTicketWorkflow. Subsequently, all calls to the microservices in the system were individually mapped into different activities. The following demonstrates the business process after our mapping strategy, also demonstrated in Figure 4.2

1. When the controller receives a request for canceling a ticket, it initiates a cancel ticket workflow.
2. Cancel ticket workflow calls to setOrderCancellingActivitiy which triggers a call to the order service to set the order to canceling
3. Cancel ticket workflow then triggers the drawBackMoneyActivity which refunds the money by internally calling inside-payment-service.
4. The cancel ticket workflow lastly calls the setOrderCancelledActivity, which internally calls order service.

4.2.2.5. Temporalint

We further wrote a simple linter [31] for Temporal to assist the experience in developing. This was possible as a result of workflows originating centrally from a single coordinator. The linter can detect a whole range of patterns and anti-patterns. Our effort in building this tool has made us more familiar with the core implementation of the Temporal framework as well as developer interactions and usage patterns of the framework. It verifies that the workflow is free from common pitfalls that might occur in writing Temporal code and notifies the end user of what needs to be done in case an issue with the input code is detected. Examples of pitfalls include the violation of deterministic rules that are essential for the framework to function properly. The following are the main capabilities of the linter.

- *Parsing:* We first build an AST of the source which allows us to inspect the tree by helping us to walk the nodes one by one.
- *Identifying Potential Fault Locations:* Before we move on to looking for violations, we need to identify potential fault locations. Since the body of the workflow is where our deterministic constraints can be violated, we identify all the workflow definitions inside the Temporal project. While writing workflows in Temporal, the workflow signature must have a special 'workflow.Context' first parameter. We navigate around the syntax tree in this stage to identify nodes containing workflow definitions.

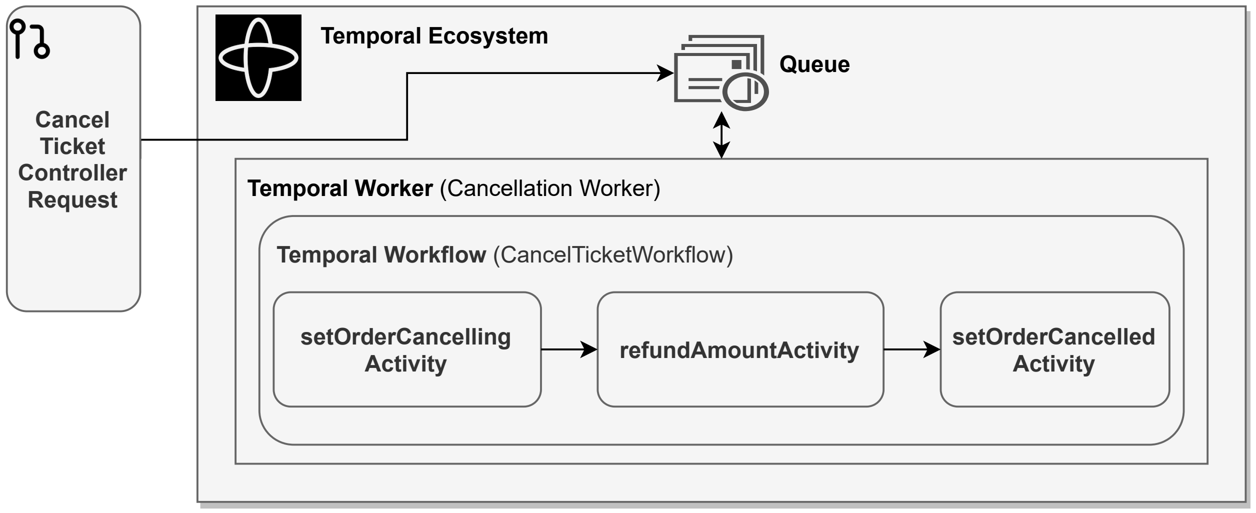

Cancel Ticket Controller Request
Temporal Ecosystem
Queue
Temporal Worker (Cancellation Worker)
Temporal Workflow (CancelTicketWorkflow)
setOrderCancelling Activity
refundAmountActivity
setOrderCancelled Activity

Figure 4.2. CancelTicket business process mapped as a Temporal workflow.

- *Finding Violations:* We look at all of the potential fault areas and verify that the workflow definition code is free from common pitfalls. If there is a violation in any workflow code, we notify the user that they need to rectify their mistake.

The orchestrated approach using Temporal orchestrated workflows allows for centralizing the processes. Thus code duplication is minimalized resulting in little to no duplication. Moreover, handling of failures is implicitly done by Temporal itself relieving the developer of having to write code to enable fault tolerance and fault resilience.

4.3. Technical Debt

Technical debt significantly impacts developer experience as developers have to work around this debt or fix it later in the process. Although technical debt (TD) may be of many types such as Architectural TD, Requirements TD, Design TD, and Test TD. Our study is scoped around Code TD which is procured as a result of a violation of coding standards and the introduction of code-smells [25]. To measure technical debt procured in the process of implementing either pattern, we follow a survey-based approach. The subject under study in this survey is a developer that has experience with the system and is familiar with the code base for orchestrated as well as the choreographed system.

First, we identify common microservice code smells which are major causes of code technical debt. We then follow a weight-based methodology where the user in our study is asked to assign weight to the likelihood of the presence of a code smell from the options including 'Not likely', 'Somewhat likely', and 'Very likely' with assigned weights 3, 2, and 1 respectively. The developer is then asked to fill out this survey after they had worked on both the orchestrated and choreographed versions of the benchmark system. Finally, we compute the overall score in either approach where a higher score means more promising results and thus the system incurs less technical debt as a result of code smells. We present the scores for orchestration and choreography in Chapter 5.

We identified the most common code smells in microservice-based systems [47] as:-

- **Cyclic Dependency**: Existence of a dependency cycle between two or more microservices.
- **Hard Coded Endpoints**: Existence of hard-coded IPs and Ports of one microservice in another.
- **Inappropriate Service Intimacy**: One microservice tries to connect and access the private data of another microservice.
- **Lack of API Gateway**: Microservices communicating with each other directly.
- **Shared Libraries**: Multiple microservices share same libraries.
- **Shared Persistence**: Different microservices attempt to access the same database.
- **Too Many Standards**: There are multiple protocols, languages, and frameworks used for the development of different microservices.
- **Duplication of Code**: Same business is re-implemented in multiple microservices.

4.4. Debugging Methodology

As an outcome of the implementation study in the previous steps, we end up with the benchmark system TrainTicket mapped in both Choreography and Orchestration. To debug the benchmark system in both patterns, the benchmark system is injected with 22 fault cases, gathered from an extensive industrial survey [56]. We then collect the total time required to debug each fault.

- **F1**: Lack of sequence control in asynchronous message delivery in cancelTicket process
- **F2**: Network congestion in ticket reservation causes delivery of requests in the wrong order
- **F3**: Requests occupy larger memory than the available resources causing service unavailability
- **F4**: SSL offloading in each microservice causes prolonged response time
- **F5**: Incoming requests to basic info service exceed the available thread pool size causing a timeout
- **F6**: Recursive errors in voucher service results in a large number of retries leading to a time out
- **F7**: Call to charge amount during the ticket purchase process intermittently times out
- **F8**: Missing request tokens during the ticket reservation process for VIP users leads to failure
- **F9**: Words on UI have incorrect display alignment
- **F10**: Ticket reservation process makes incorrect API calls resulting in failure
- **F11**: A missing edge case causes intermittent lack of sequence control during the cancellation process resulting in failure
- **F12**: A cancellation request to the order service for a locked station rejects any incoming requests resulting in failure
- **F13**: A transmission delay in simultaneous requests by the same user over a short period makes the system inconsistent
- **F14**: Calculation of the price of a seat is wrong
- **F15**: A lengthy request body size results in Nginx to block the request

- **F16**: Adding routes by file upload with a size bigger than the limit results in the rejection of the file
- **F17**: Requesting voucher with a simulated load delay in SQL results in a query timeout
- **F18**: A missing null value check during the train selection process results in an error in the getFood response
- **F19**: Display of package consignment prices in French is in the wrong format
- **F20**: Mismatch in the version of a common library versioning results in loading different versions of the same data structure
- **F21**: Missing aria-labeled-by in verification code field in login results in poor accessibility
- **F22**: A mismatch of the column name in the select and from part of an SQL query results in empty results during voucher printing

After mapping the entire system to Temporal, we organize the source code on our GitHub repository [30] on different branches in such a way that a user can preview what changes have to be made to the existing system to map the business flow to Temporal and changes to the source to inject the faults above.

For example, to be able to see how fault F1 is mapped onto Temporal, one only needs to do a diff from master to F1 to understand the source code changes. An example of this diff can be found in the appendix.

Consistent with the benchmark system [56], the 22 fault cases are subdivided into three types.

1. **Internal**: Result of implementation in microservice code.
2. **Interaction**: Result of interaction between microservices.
3. **Environment**: Result of external infrastructure and misconfiguration.

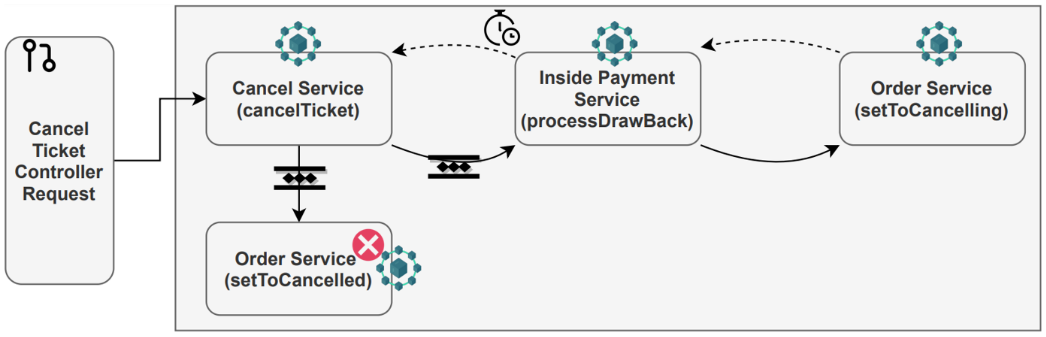

Cancel Ticket Controller Request
Cancel Service (cancelTicket)
Inside Payment Service (processDrawBack)
Order Service (setToCancelling)
Order Service (setToCancelled)

Figure 4.3. Original cancelTicket process with fault injected. Error is thrown while setting the order to cancelled.

4.4.1. Choreography

As the original benchmark system is a choreographed version, we utilize the time to debug the faults in the original study as our baseline for the choreographed approach and compare these to the ones in the orchestrated approach. We discuss this comparison in Chapter 5.

4.4.2. Orchestration

In the orchestrated scheme of TrainTicket, built using Temporal, we employ tools provided by Temporal itself to analyze and debug the 22 fault cases. To study this debugging process, we consider the previously highlighted cancelTicket process with the injected fault, demonstrated in Figure 4.3. In TrainTicket, this fault is injected by making the system lack strict sequence control and simulating network congestion while calling the inside payment service. Upon simulating the network congestion, the refund call gets triggered after the order service already sets the order to canceled which throws an error as only active orders can be further processed for a refund. In Temporal, we invoke drawBackMoneyActivity and setOrderCancelledAcitivty asynchronously to replicate the fault in the cancelTicket workflow. Further, we highlight the steps taken towards debugging it from the Temporal system with the help of a formal model [24].

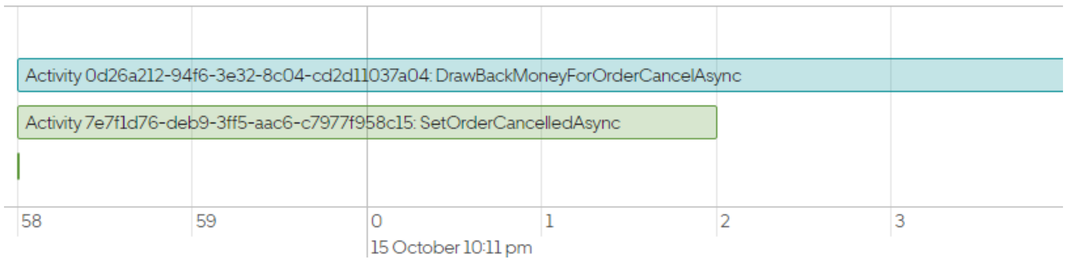
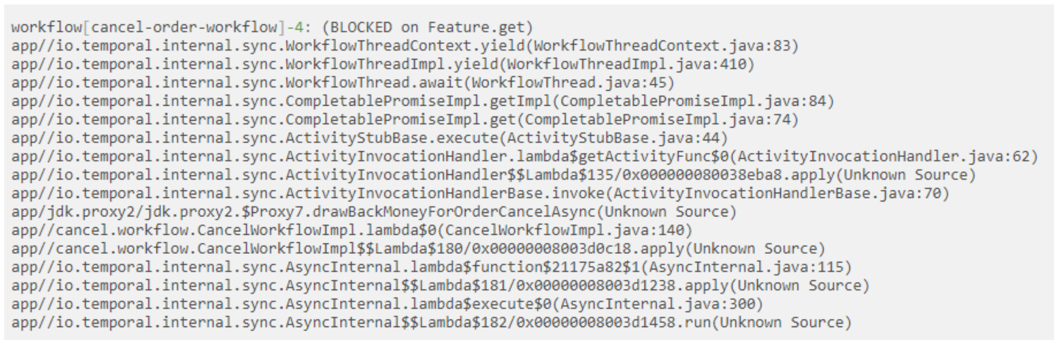


Figure 4.4. Visual trace of faulty cancelTicket workflow.



```
workflow[cancel-order-workflow]-4: (BLOCKED on Feature.get)
app//io.temporal.internal.sync.WorkflowThreadContext.yield(WorkflowThreadContext.java:83)
app//io.temporal.internal.sync.WorkflowThreadImpl.yield(WorkflowThreadImpl.java:410)
app//io.temporal.internal.sync.WorkflowThread.await(WorkflowThread.java:45)
app//io.temporal.internal.sync.CompletablePromiseImpl.getImpl(CompletablePromiseImpl.java:84)
app//io.temporal.internal.sync.CompletablePromiseImpl.get(CompletablePromiseImpl.java:74)
app//io.temporal.internal.sync.ActivityStubBase.execute(ActivityStubBase.java:44)
app//io.temporal.internal.sync.ActivityInvocationHandler.lambda$getActivityFunc$0(ActivityInvocationHandler.java:62)
app//io.temporal.internal.sync.ActivityInvocationHandler$$Lambda$135/0x000000080038eba8.apply(Unknown Source)
app//io.temporal.internal.sync.ActivityInvocationHandlerBase.invoke(ActivityInvocationHandlerBase.java:70)
app/jdk.proxy2/jdk.proxy2.$Proxy7.drawBackMoneyForOrderCancelAsync(Unknown Source)
app//cancel.workflow.CancelWorkflowImpl.lambda$0(CancelWorkflowImpl.java:140)
app//cancel.workflow.CancelWorkflowImpl$$Lambda$180/0x00000008003d0c18.apply(Unknown Source)
app//io.temporal.internal.sync.AsyncInternal.lambda$function$21175a82$1(AsyncInternal.java:115)
app//io.temporal.internal.sync.AsyncInternal$$Lambda$181/0x00000008003d1238.apply(Unknown Source)
app//io.temporal.internal.sync.AsyncInternal.lambda$execute$0(AsyncInternal.java:300)
app//io.temporal.internal.sync.AsyncInternal$$Lambda$182/0x00000008003d1458.run(Unknown Source)
```

Figure 4.5. Stack trace of faulty cancelTicket workflow.

- *Constructing Problem Space*: In this step, we develop an initial understanding of the fault. The Temporal Web UI tool produces a stack trace. We collect these to develop fault perception. The stack trace and visual trace for the cancelTicket workflow are demonstrated in Figure 4.5 and Figure 4.4 respectively.
- *Fault Symptom Identification*: In this step, we set the environment to be able to reproduce the fault. As a benefit of Temporal, the workflow made the business process explicit and we were able to set a minimalistic version of the environment by quickly identifying and deploying only the microservices involved in the workflow.
- *Diagnosis*: In this step, we used the knowledge gained through the previous steps to predict and hypothesize the locality of the fault. From careful observation of the visual trace, we found that setOrderCancelledActivity invokes before drawBackMoney-ForCancelActivity, which breaks the routine business expectation.

- *Solution Generation*: The final step was correcting the fault. In this case, this was achieved by adding the necessary sequence control. After the correction, we ran multiple test cases to verify the correctness of the solution.

5. RESULTS AND DISCUSSION

This section highlights the results of our study and further discusses the findings concerning the original research questions. The user in our study had more than three years of industry experience in developing microservices.

5.1. Results

5.1.1. Tools and Support

During the case study, the developers were able to find more tools for orchestration-based approaches, in comparison to microservice choreography. While tools like istio service mesh enabled monitoring, there were no framework-based methodologies towards developing choreographed system. Whereas, in Microservice Orchestration we found several frameworks that handled low level distributed tasks.

We believe that this difference in the availability of tools is a result of the autonomous nature of microservice in a choreography-based approach. Since every microservice in choreography is independent of each other, it is hard to conform every microservice to a uniform set of rules or patterns that is the basic assumptions of such a framework. Whereas, an orchestrator is meant to have more control over the system as it coordinates the entire communication, enabling the possibility of add-ons such as frameworks on this central layer which is the brain of the system.

5.1.2. Technical Debt

We used a negative weighing methodology where a developer scores higher points if there is a low likelihood of the presence of an issue in code and lower points for a high likelihood, hence an overall high score is desirable. Table 5.1 and Table 5.2 highlight the results for choreography and orchestrated versions of the system respectively.

Out of the highest point mark of 27, the choreographed system achieves a total score of 12 points whereas orchestrated system scores 20 points. This indicates that the orchestrated

Table 5.1. Technical debt survey for choreography based benchmark system implementation

Code Smell	Not Likely (3)	Somewhat Likely (2)	Very Likely (1)
Did the code have cyclic dependencies between microservices?		✓	
Is there a presence of hard coded endpoints?			✓
Does one microservice try to access private data of another?	✓		
Are microservices communicating directly?			✓
What is the likely hood of the existence of shared libraries?			✓
Does the system have shared persistence?		✓	
Does the code follow too many different standards?			✓
What was the likely hood of code duplication?			✓

Table 5.2. Technical debt survey for orchestration based benchmark system implementation

Code Smell	Not Likely (3)	Somewhat Likely (2)	Very Likely (1)
Did the code have cyclic dependencies between microservices?	✓		
Is there a presence of hard coded endpoints?			✓
Does one microservice try to access private data of another?		✓	
Are microservices communicating directly?	✓		
What is the likely hood of the existence of shared libraries?	✓		
Does the system have shared persistence?		✓	
Does the code follow too many different standards?	✓		
What was the likely hood of code duplication?	✓		

version of the system has a lower overall Code TD (Technical Debt) whereas, the choreographed approach procures more Code TD.

The presence of high code technical debt is a negative in terms of developer experience as the system becomes complex and hard to maintain over time. This means that microservice orchestration is more favorable for a better developer experience.

5.1.3. Debugging

Table 5.3 sums up our findings of the time required to debug the faults in the orchestrated system, to the time required to debug those faults in the original choreography-based system. Temporal did not play a significant role in helping the faults that were internal as these faults are only a result of miscommunicated requirements. Whereas, on the occurrence of an interactional fault causing exceptions, Temporal handled fault tolerance and preserved the entire state of the program including local variables, inputs, and outputs. The function resumed as soon as the fault was corrected. Similarly, for environment faults, the execution resumed automatically as soon as the microservice came back live. This was not the case in the orches-

Table 5.3. Comparison of debugging times between original study and system mapped into Temporal

Fault	Type	Overall Time (H)	Overall Time Temporal (H)
Fl	Interaction	13.6	2.4
F2	Interaction	13.9	3.6
F3	Environment	Failed	0.5
F4	Environment	Failed	0.4
F5	Interaction	12.6	5.2
F6	Interaction	5.9	6
F7	Interaction	12	5.3
F8	Interaction	12.2	4.8
F9	Internal	1.8	0.9
F10	Interaction	10.6	6.1
F11	Interaction	13.9	4.8
F12	Interaction	19.3	8.1
F13	Interaction	16	7.6
F14	Internal	2.9	3
F15	Environment	1.8	0.6
F16	Environment	5.9	6.1
F17	Internal	5.9	4.1
F18	Internal	3.4	3.9
F19	Internal	0.7	0.7
F20	Environment	3.8	3.1
F21	Internal	1.6	1.5
F22	Internal	0.4	0.5

trated approach, as not every place had explicit error handling resulting in the microservices crashing and the whole operation being lost, having to start over.

As highlighted in Figure 5.1, it is evident that the time required to debug the faults in a Temporal based microservice orchestration-based approach was way less than the time that was required to debug these faults in the original choreographed version of the benchmark system. Hence improving the overall speed of debugging.

5.2. Discussion

5.2.1. Reality of Choreography

Microservice choreography sounds very promising for agility and lose coupling. It implies that services are simply connected or disconnected from the broker. However, teams that have built complex systems shared that as a result of the growth of the system, process flows were embedded within the source code of multiple services, which is troublesome for mainte-

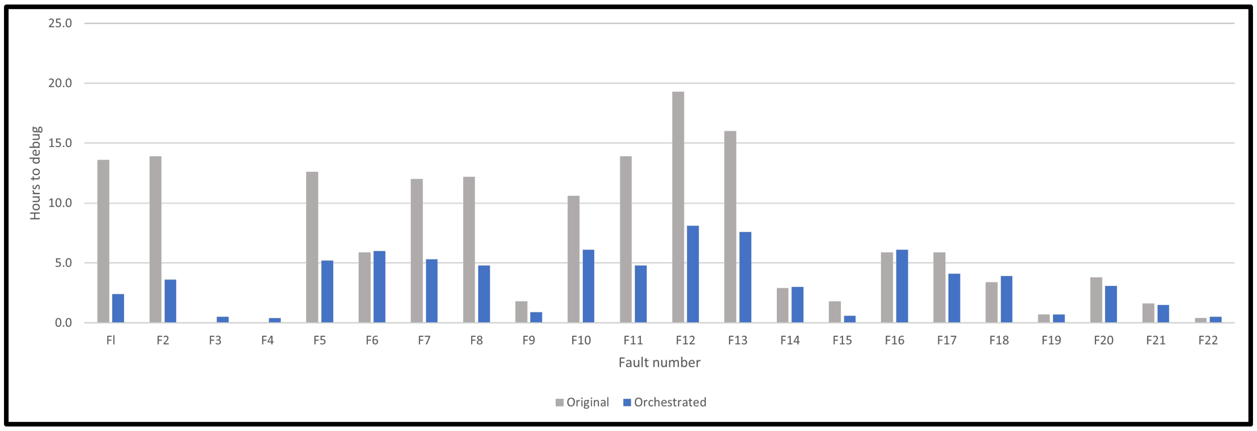


Figure 5.1. Barchart of times taken to debug choreographed vs orchestrated approach.

nance activities including debugging. In addition to that, it is hard to monitor these services and it becomes very complex to answer the current progress of an initiated task.

With the continuous addition of microservices in a choreographed system, the communications become a dependency hell, an example of which can be seen in Figure 5.2.

5.2.2. Promise of Orchestration and Workflow Engines

We believe that employing a workflow engine-based approach for the implementation of the choreography pattern was fruitful in multiple ways. Similar to the early days of computing, before SQL was invented as a standard for querying and saving data, everyone invented their storage and retrieving mechanism. Workflow engines such as Temporal, similarly handle the low-level distributed tasks to how SQL does for data and we envision that these engines have the potential to become broiler plates for every microservice-based system.

5.2.3. Testing and Debugging Using Temporal

Temporal provides a test framework to facilitate testing Workflow implementations. The framework supports unit tests as well as functional tests of the Workflow logic. Temporal provides a Web UI that can be used to view Workflow Execution states or explore and debug Workflow Executions. Each workflow can be explored based on its ID and argument, variables and other parts of state and history are also fully available. There is no need to use any external

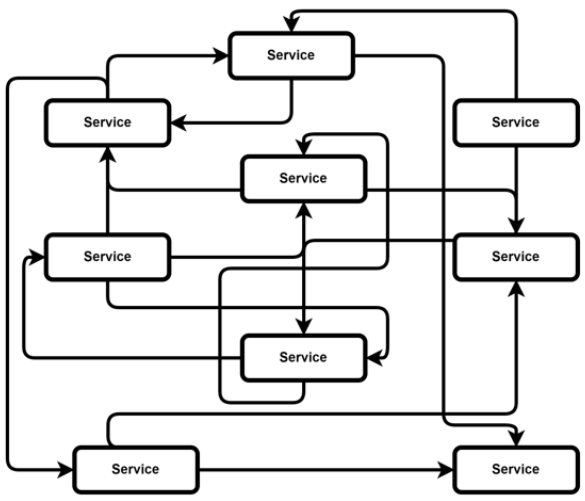


Figure 5.2. A result of dependencies created by increasing microservices in choreography.

tool for debugging. In comparison to the original microservices-benchmark work [56] where the authors had to use external tools in their empirical study of debugging process.

6. CONCLUSION

In this work, we have presented a detailed literature review in which we identified the main concepts including monolithic architecture and its challenges, the advantages of microservices, and the patterns of microservice communication including choreography and orchestration. We conducted a study to compare the orchestration and choreography patterns. First, we highlighted results from our study and experience of developing a benchmark system in both patterns where we compared the available tools and technical debt procured in both approaches. Then we compared the time required to debug the system in both approaches. Through thorough research and study, we were able to achieve results in the favor of microservice orchestration making the program easier to debug, producing less technical debt, and having the availability of more tools than choreography. Our experiment is designed to be scalable and can be easily extended for a larger study group.

From the findings of our study, we were able to answer our research questions as highlighted in Chapter 1.

- **RQ1: How does the choice of the interaction pattern affect the implementation process of the systems?** The choice of pattern affects the system in multiple ways. A thorough literature review led us to hypothesize that selecting a specific pattern comes with its own pros and cons. We compared this implementation aspect as a measure of quality, in terms of the produced technical debt and the availability of tools and support of either pattern.
- **RQ2: What are the immediate pros and cons of selecting either of the approaches?** When selecting an appropriate pattern, there are multiple immediate pros and cons associated including factors that come into play immediately. During our study, we found multiple benefits of using the orchestrated approach that started from the development phase in terms of handling low-level distributed tasks until debugging, providing us

with tools off the shelf to assist the maintenance phase as well. Whereas, while working with choreography we faced a lack of tool support.

- **RQ3: How does the choice affect the process of debugging the system?** The choice of an appropriate pattern has a great impact on post-development activities such as debugging in both the development and the maintenance phase. From Table 5.3, it can be seen that it took us a shorter amount of time to debug a microservice orchestration-based system in comparison to a microservice choreography-based system.

REFERENCES

[1] Omar Al-Debagy and Péter Martinek. Extracting microservices' candidates from monolithic applications: interface analysis and evaluation metrics approach. In *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*, pages 289–294. IEEE, 2020.

[2] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[3] Apache. Apache airflow. Available: https://airflow.apache.org, 2022.

[4] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[5] Saša Baškarada, Vivian Nguyen, and Andy Koronios. Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems*, 60(5):428–436, 2020.

[6] Terese Besker, Antonio Martini, and Jan Bosch. Technical debt cripples software developer productivity: a longitudinal study on developers' daily software development work. In *Proceedings of the 2018 International Conference on Technical Debt*, pages 105–114, 2018.

[7] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering*, 26(5):1–39, 2021.

[8] Stephen K Burley, Helen M Berman, Charmi Bhikadiya, Chunxiao Bi, Li Chen, Luigi Di Costanzo, Cole Christie, Ken Dalenberg, Jose M Duarte, Shuchismita Dutta, et al. Rcsb protein data bank: biological macromolecular structures enabling research and ed-

ucation in fundamental biology, biomedicine, biotechnology and energy. *Nucleic acids research*, 47(D1):D464–D474, 2019.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[10] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6, 2016.

[11] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: Current and future directions. 17(4):29–45, jan 2018.

[12] Lan Cheng, Emerson Rex Murphy-Hill, Mark Canning, Ciera Nicole Christopher Jaspan, Collin Green, Andrea Marie Knight Dolan, Nan Zhang, and Elizabeth Kammer. What improves developer productivity at google? code quality. In *Foundations of Software Engineering: Industry Paper*, 2022.

[13] Cesar de la Torre, Bill Wagner, and Mike Rousos. .net microservices: Architecture for containerized .net applications. *Microsoft Developer Division*, 2020.

[14] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96, 2019.

[15] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[16] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30, 2017.

[17] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[18] Nishant Garg. *Apache kafka*. Packt Publishing Birmingham, UK, 2013.

[19] Mahtab Haj Ali. *Measuring the Modeling Complexity of Microservice Choreography and Orchestration: The Case of E-commerce Applications*. PhD thesis, Université d'Ottawa/University of Ottawa, 2021.

[20] Wilhelm Hasselbring and Guido Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, 2017.

[21] Marcus Hilbrich and Fabian Lehmann. Discussing microservices: Definitions, pitfalls, and their relations. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 39–44. IEEE, 2022.

[22] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[23] Ravi Jhawar and Vincenzo Piuri. Fault tolerance and resilience in cloud computing environments. In *Computer and information security handbook*, pages 165–181. Elsevier, 2017.

[24] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, ACE '19, page 79–86, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[26] Antonios Makris, Konstantinos Tserpes, and Theodora Varvarigou. Transition from monolithic to microservice-based applications. challenges from the developer perspective. *Open Research Europe*, 2(24):24, 2022.

[27] Samar Abbas Maxim Fateev. Temporal workflow engine. Available: https://temporal.io, 2022.

[28] Alan Megargel, Christopher M. Poskitt, and Venky Shankararaman. Microservices orchestration vs. choreography: A decision framework. In *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, pages 134–141, 2021.

[29] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2, 2014.

[30] Anas Nadeem. Trainticket-workflows. Available: https://github.com/ansnadeem/train ticket-workflows, 2022.

[31] Anas Nadeem and Muhammad Zubair Malik. Temporalint: A linter for temporal.io. Available:https://github.com/ansnadeem/temporalint, 2020.

[32] Anas Nadeem and Muhammad Zubair Malik. A case for microservices orchestration using workflow engines. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '22, page 6–10, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Netflix. Conductor workflow engine. Available: https://netflix.github.io/conductor/, 2021.

[34] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[35] Chris Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[36] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019.

[37] RV Rajesh. *Spring Microservices*. Packt Publishing Ltd, 2016.

[38] Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze. Highly-available applications on unreliable infrastructure: Microservice architectures in practice. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 130–137, 2017.

[39] Chaitanya K Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8), 2018.

[40] Adalberto R. Sampaio, Harshavardhan Kadiyala, Bo Hu, John Steinbacher, Tony Erwin, Nelson Rosa, Ivan Beschastnikh, and Julia Rubin. Supporting microservice evolution. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 539–543, 2017.

[41] Vindeep Singh and Sateesh K Peddoju. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852, 2017.

[42] Neha Singhal, Usha Sakthivel, and Pethuru Raj. Selection mechanism of micro-services orchestration vs. choreography. *International Journal of Web & Semantic Technology (IJWesT)*, 10(1):25, 2019.

[43] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in action*, volume 47. Manning Greenwich Conn., 2011.

[44] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[45] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native docker clustering with swarm*. Packt Publishing Ltd, 2016.

[46] Tatjana D Stojanovic, Sasa D Lazarevic, Milos Milic, and Ilija Antovic. Identifying microservices using structured system analysis. In *2020 24th International Conference on Information Technology (IT)*, pages 1–4. IEEE, 2020.

[47] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.

[48] Uber. Uber cadence workflow engine. Available: https://cadenceworkflow.io/, 2022.

[49] Pedro Valderas, Victoria Torres, and Vicente Pelochano. Supporting a hybrid composition of microservices. the eucaliptool platform. *Journal of Software Engineering Research and Development*, 8:1–1, 2020.

[50] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015.

[51] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 179–182, 2016.

[52] Shangguang Wang, Yan Guo, Ning Zhang, Peng Yang, Ao Zhou, and Xuemin Shen. Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach. *IEEE Transactions on Mobile Computing*, 20(3):939–951, 2019.

[53] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software*, 182:111061, 2021.

[54] Qilin Xiang, Xin Peng, Chuan He, Hanzhang Wang, Tao Xie, Dewei Liu, Gang Zhang, and Yuanfang Cai. No free lunch: Microservice practices reconsidered in industry, 2021.

[55] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*, 2019.

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.

[57] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Wenhai Li, Chao Ji, and Dan Ding. Delta debugging microservice systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 802–807. IEEE, 2018.

[58] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In Michel Chaudron,

Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM, 2018.

APPENDIX

```
diff --git a/ts-cancel-service/src/main/java/cancel/async/AsyncTask.
    ↪ java b/ts-cancel-service/src/main/java/cancel/async/AsyncTask.
    ↪ java
index 98e5d6e..ae66130 100644
--- a/ts-cancel-service/src/main/java/cancel/async/AsyncTask.java
+++ b/ts-cancel-service/src/main/java/cancel/async/AsyncTask.java
@@ -1,63 +1,78 @@
-package cancel.async;
-
-import java.util.concurrent.Future;
-import cancel.domain.*;
-import org.slf4j.Logger;
-import org.slf4j.LoggerFactory;
-import org.springframework.beans.factory.annotation.Autowired;
-import org.springframework.scheduling.annotation.Async;
-import org.springframework.scheduling.annotation.AsyncResult;
-import org.springframework.stereotype.Component;
-import org.springframework.web.client.RestTemplate;
-
-
-
-@Component
-public class AsyncTask {
-    protected final Logger logger = LoggerFactory.getLogger(this.getClass
    ↪ ());
-
```

```
-    @Autowired
-        private RestTemplate restTemplate;
-
-    @Async("myAsync")
-    public Future<ChangeOrderResult> updateOtherOrderStatusToCancel(
↪ ChangeOrderInfo info) throws InterruptedException{
-
-        Thread.sleep(2000);
-
-        System.out.println("[Cancel␣Order␣Service][Change␣Order␣Status]␣
↪ Getting....");
-        ChangeOrderResult result = restTemplate.postForObject("http://ts-
↪ order-other-service:12032/orderOther/update",info,ChangeOrderResult.
↪ class);
-        return new AsyncResult<>(result);
-    }
-
-    @Async("mySimpleAsync")
-    public Future<Boolean> drawBackMoneyForOrderCan(String money, String
↪ userId,String orderId) throws InterruptedException{
-
-        System.out.println("[Cancel␣Order␣Service][Get␣Order]␣Getting....
↪ ");
-        GetOrderByIdInfo getOrderInfo = new GetOrderByIdInfo();
-        getOrderInfo.setOrderId(orderId);
-        GetOrderResult cor = restTemplate.postForObject(
-                "http://ts-order-other-service:12032/orderOther/getById/"
-                ,getOrderInfo,GetOrderResult.class);
-           Order order = cor.getOrder();
-           if(order.getStatus() == OrderStatus.NOTPAID.getCode()
-                   || order.getStatus() == OrderStatus.PAID.getCode() ||
    ↪ order.getStatus() == OrderStatus.CHANGE.getCode()){
-
-               System.out.println("[Cancel␣Order␣Service][Draw␣Back␣Money]␣
    ↪ Draw␣back␣money...");
-               DrawBackInfo info = new DrawBackInfo();
-               info.setMoney(money);
-               info.setUserId(userId);
-               String result = restTemplate.postForObject("http://ts-inside-
    ↪ payment-service:18673/inside_payment/drawBack",info,String.class);
-               if(result.equals("true")){
-                   return new AsyncResult<>(true);
-               }else{
-                   return new AsyncResult<>(false);
-               }
-           }else{
-
-               System.out.println("[Cancel␣Order␣Service][Drawback␣Money]␣
    ↪ Fail.␣Status␣Not␣Permitted");
-               return new AsyncResult<>(false);
-
-           }
-       }
-
-}
+package cancel.async;
```

```
+
+import java.util.Random;
+import java.util.concurrent.Future;
+import cancel.domain.*;
+import org.springframework.beans.factory.annotation.Autowired;
+import org.springframework.scheduling.annotation.Async;
+import org.springframework.scheduling.annotation.AsyncResult;
+import org.springframework.stereotype.Component;
+import org.springframework.web.client.RestTemplate;
+
+
+
+@Component
+public class AsyncTask {
+
+        //private String hostname = "172.25.234.77";
+
+    @Autowired
+        private RestTemplate restTemplate = new RestTemplate();
+
+    @Async("myAsync")
+    public Future<ChangeOrderResult> updateOtherOrderStatusToCancel(
   ↪ ChangeOrderInfo info) throws InterruptedException{
+
+         Thread.sleep(4000);
+
+         System.out.println("[Cancel␣Order␣Service][Change␣Order␣Status]")
   ↪ ;
```

```
+           ChangeOrderResult result = restTemplate.postForObject("http://ts-
    ↪ order-other-service:12032/orderOther/update",info,ChangeOrderResult.
    ↪ class);
+           return new AsyncResult<>(result);
+
+       }
+
+       @Async("mySimpleAsync")
+       public Future<Boolean> drawBackMoneyForOrderCancel(String money,
    ↪ String userId,String orderId, String loginToken) throws
    ↪ InterruptedException{
+
+           /*********************** Fault Reproduction - Error Process Seq
    ↪ ************************/
+           //double op = new Random().nextDouble();
+           //if(op < 1.0){
+           //     System.out.println("[Cancel Order Service] Delay
    ↪ Process Wrong Cancel Process");
+                Thread.sleep(8000);
+           //} else {
+           //     System.out.println("[Cancel Order Service] Normal
    ↪ Process Normal Cancel Process");
+           //}
+
+
+           //1.Search Order Info
+           System.out.println("[Cancel␣Order␣Service][Get␣Order]␣Getting....
    ↪ ");
```

```
+           GetOrderByIdInfo getOrderInfo = new GetOrderByIdInfo();
+           getOrderInfo.setOrderId(orderId);
+           GetOrderResult cor = restTemplate.postForObject(
+                   "http://ts-order-other-service:12032/orderOther/getById/"
+                   ,getOrderInfo,GetOrderResult.class);
+           Order order = cor.getOrder();
+           System.out.println("[Cancel␣Order␣Service]Got␣order␣successfully"
    ↪ );
+
+           //2.Change order status to cancelling
+           order.setStatus(OrderStatus.Canceling.getCode());
+           ChangeOrderInfo changeOrderInfo = new ChangeOrderInfo();
+           changeOrderInfo.setOrder(order);
+           changeOrderInfo.setLoginToken(loginToken);
+           ChangeOrderResult changeOrderResult = restTemplate.postForObject(
    ↪ "http://ts-order-other-service:12032/orderOther/update",
    ↪ changeOrderInfo,ChangeOrderResult.class);
+           if(changeOrderResult.isStatus() == false){
+               System.out.println("[Cancel␣Order␣Service]Unexpected␣error");
+           }
+           //3.do drawback money
+           System.out.println("[Cancel␣Order␣Service][Draw␣Back␣Money]␣Draw␣
    ↪ back␣money...");
+           DrawBackInfo info = new DrawBackInfo();
+           info.setMoney(money);
+           info.setUserId(userId);
+           String result = restTemplate.postForObject("http://ts-inside-
    ↪ payment-service:18673/inside_payment/drawBack",info,String.class);
```

```
+           if(result.equals("true")){
+               return new AsyncResult<>(true);
+           }else{
+               return new AsyncResult<>(false);
+           }
+           /***********/
+       }
+}
diff --git a/ts-cancel-service/src/main/java/cancel/controller/
    ↪ CancelController.java b/ts-cancel-service/src/main/java/cancel/
    ↪ controller/CancelController.java
index 3eb5887..1bbe0e3 100644
--- a/ts-cancel-service/src/main/java/cancel/controller/CancelController.
    ↪ java
+++ b/ts-cancel-service/src/main/java/cancel/controller/CancelController.
    ↪ java
@@ -1,70 +1,70 @@
-package cancel.controller;
-
-import cancel.domain.CalculateRefundResult;
-import cancel.domain.CancelOrderInfo;
-import cancel.domain.CancelOrderResult;
-import cancel.domain.VerifyResult;
-import cancel.service.CancelService;
-import org.springframework.beans.factory.annotation.Autowired;
-import org.springframework.web.bind.annotation.*;
-import org.springframework.web.client.RestTemplate;
-
```

```
@RestController
public class CancelController {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    CancelService cancelService;

    @CrossOrigin(origins = "*")
    @RequestMapping(path = "/cancelCalculateRefund", method =
  ↪ RequestMethod.POST)
    public CalculateRefundResult calculate(@RequestBody CancelOrderInfo
  ↪ info){
        System.out.println("[Cancel␣Order␣Service][Calculate␣Cancel␣
  ↪ Refund]␣OrderId:" + info.getOrderId());
        return cancelService.calculateRefund(info);
    }

    @CrossOrigin(origins = "*")
    @RequestMapping(path = "/cancelOrder", method = RequestMethod.POST)
    public CancelOrderResult cancelTicket(@RequestBody CancelOrderInfo
  ↪ info, @CookieValue String loginToken, @CookieValue String loginId){
        System.out.println("[Cancel␣Order␣Service][Cancel␣Ticket]␣info:"
  ↪ + info.getOrderId());
        if(loginToken == null ){
            loginToken = "admin";
        }
```

```
-         System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣order␣ID
   ↪ :" + info.getOrderId() + "␣␣loginToken:" + loginToken);
-         if(loginToken == null){
-             System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣Not␣
   ↪ receive␣any␣login␣token");
-             CancelOrderResult result = new CancelOrderResult();
-             result.setStatus(false);
-             result.setMessage("No␣Login␣Token");
-             return result;
-         }
-         VerifyResult verifyResult = verifySsoLogin(loginToken);
-         if(verifyResult.isStatus() == false){
-             System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣Do␣
   ↪ not␣login.");
-             CancelOrderResult result = new CancelOrderResult();
-             result.setStatus(false);
-             result.setMessage("Not␣Login");
-             return result;
-         }else{
-             System.out.println("[Cancel␣Order␣Service][Cancel␣Ticket]␣
   ↪ Verify␣Success");
-             try{
-                 return cancelService.cancelOrder(info,loginToken,loginId)
   ↪ ;
-             }catch(Exception e){
-                 e.printStackTrace();
-                 return null;
-             }
-
-            }
-        }
-
-        private VerifyResult verifySsoLogin(String loginToken){
-            System.out.println("[Cancel␣Order␣Service][Verify␣Login]␣
    ↪ Verifying....");
-            VerifyResult tokenResult = restTemplate.getForObject(
-                    "http://ts-sso-service:12349/verifyLoginToken/" +
    ↪ loginToken,
-                    VerifyResult.class);
-            return tokenResult;
-        }
-
-}
+package cancel.controller;
+
+import cancel.domain.CalculateRefundResult;
+import cancel.domain.CancelOrderInfo;
+import cancel.domain.CancelOrderResult;
+import cancel.domain.VerifyResult;
+import cancel.service.CancelService;
+import org.springframework.beans.factory.annotation.Autowired;
+import org.springframework.web.bind.annotation.*;
+import org.springframework.web.client.RestTemplate;
+
+@RestController
+public class CancelController {
```

```
+
+     @Autowired
+     private RestTemplate restTemplate;
+
+     @Autowired
+     CancelService cancelService;
+
+     @CrossOrigin(origins = "*")
+     @RequestMapping(path = "/cancelCalculateRefund", method =
   ↪ RequestMethod.POST)
+     public CalculateRefundResult calculate(@RequestBody CancelOrderInfo
   ↪ info){
+         System.out.println("[Cancel␣Order␣Service][Calculate␣Cancel␣
   ↪ Refund]␣OrderId:" + info.getOrderId());
+         return cancelService.calculateRefund(info);
+     }
+
+     @CrossOrigin(origins = "*")
+     @RequestMapping(path = "/cancelOrder", method = RequestMethod.POST)
+     public CancelOrderResult cancelTicket(@RequestBody CancelOrderInfo
   ↪ info, @CookieValue String loginToken, @CookieValue String loginId)
   ↪ throws RuntimeException{
+         System.out.println("[Cancel␣Order␣Service][Cancel␣Ticket]␣info:"
   ↪ + info.getOrderId());
+         if(loginToken == null ){
+             loginToken = "admin";
+         }
+           System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣order␣ID
    ↪ :" + info.getOrderId() + "␣␣loginToken:" + loginToken);
+           if(loginToken == null){
+               System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣Not␣
    ↪ receive␣any␣login␣token");
+               CancelOrderResult result = new CancelOrderResult();
+               result.setStatus(false);
+               result.setMessage("No␣Login␣Token");
+               return result;
+           }
+           VerifyResult verifyResult = verifySsoLogin(loginToken);
+           if(verifyResult.isStatus() == false){
+               System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣Do␣
    ↪ not␣login.");
+               CancelOrderResult result = new CancelOrderResult();
+               result.setStatus(false);
+               result.setMessage("Not␣Login");
+               return result;
+           }else{
+               System.out.println("[Cancel␣Order␣Service][Cancel␣Ticket]␣
    ↪ Verify␣Success");
+               try{
+                   return cancelService.cancelOrder(info,loginToken,loginId)
    ↪ ;
+               }catch(Exception e){
+                   e.printStackTrace();
+                   return null;
+               }
+
+           }
+       }
+
+       private VerifyResult verifySsoLogin(String loginToken){
+           System.out.println("[Cancel␣Order␣Service][Verify␣Login]␣
    ↪ Verifying....");
+           VerifyResult tokenResult = restTemplate.getForObject(
+                   "http://ts-sso-service:12349/verifyLoginToken/" +
    ↪ loginToken,
+                   VerifyResult.class);
+           return tokenResult;
+       }
+
+}
diff --git a/ts-cancel-service/src/main/java/cancel/domain/OrderStatus.
    ↪ java b/ts-cancel-service/src/main/java/cancel/domain/OrderStatus.
    ↪ java
index fe9ba85..3f7632c 100644
--- a/ts-cancel-service/src/main/java/cancel/domain/OrderStatus.java
+++ b/ts-cancel-service/src/main/java/cancel/domain/OrderStatus.java
@@ -1,39 +1,40 @@
-package cancel.domain;
-
-public enum OrderStatus {
-
-       NOTPAID    (0,"Not␣Paid"),
-       PAID       (1,"Paid␣&␣Not␣Collected"),
COLLECTED (2,"Collected"),
CHANGE    (3,"Cancel␣&␣Rebook"),
CANCEL    (4,"Cancel"),
REFUNDS   (5,"Refunded"),
USED      (6,"Used");

private int code;
private String name;

OrderStatus(int code, String name){
this.code = code;
this.name = name;
}

public int getCode(){
return code;
}

public String getName() {
return name;
}

public static String getNameByCode(int code){
OrderStatus[] orderStatusSet = OrderStatus.values();
for(OrderStatus orderStatus : orderStatusSet){
if(orderStatus.getCode() == code){
    return orderStatus.getName();
}
```

```
-           }
-           return orderStatusSet[0].getName();
-       }
-
-}
+package cancel.domain;
+
+public enum OrderStatus {
+
+       NOTPAID    (0,"Not␣Paid"),
+       PAID       (1,"Paid␣&␣Not␣Collected"),
+       COLLECTED  (2,"Collected"),
+       CHANGE     (3,"Cancel␣&␣Rebook"),
+       CANCEL     (4,"Cancel"),
+       REFUNDS    (5,"Refunded"),
+       USED       (6,"Used"),
+       Canceling  (100,"Canceling");
+
+       private int code;
+       private String name;
+
+       OrderStatus(int code, String name){
+           this.code = code;
+           this.name = name;
+       }
+
+       public int getCode(){
+           return code;
+     }
+
+     public String getName() {
+         return name;
+     }
+
+     public static String getNameByCode(int code){
+         OrderStatus[] orderStatusSet = OrderStatus.values();
+         for(OrderStatus orderStatus : orderStatusSet){
+             if(orderStatus.getCode() == code){
+                 return orderStatus.getName();
+             }
+         }
+         return orderStatusSet[0].getName();
+     }
+
+}
diff --git a/ts-cancel-service/src/main/java/cancel/service/CancelService.
    ↪ java b/ts-cancel-service/src/main/java/cancel/service/CancelService.
    ↪ java
index 989cb88..05658f4 100644
--- a/ts-cancel-service/src/main/java/cancel/service/CancelService.java
+++ b/ts-cancel-service/src/main/java/cancel/service/CancelService.java
@@ -1,13 +1,41 @@
-package cancel.service;
-
-import cancel.domain.CalculateRefundResult;
-import cancel.domain.CancelOrderInfo;
```

```
-import cancel.domain.CancelOrderResult;
-
-public interface CancelService {
-
-    CancelOrderResult cancelOrder(CancelOrderInfo info,String loginToken,
   ↪ String loginId) throws Exception;
-
-    CalculateRefundResult calculateRefund(CancelOrderInfo info);
-
-}
+package cancel.service;
+
+import cancel.domain.CalculateRefundResult;
+import cancel.domain.CancelOrderInfo;
+import cancel.domain.CancelOrderResult;
+import cancel.domain.ChangeOrderInfo;
+import cancel.domain.ChangeOrderResult;
+import cancel.domain.GetAccountByIdInfo;
+import cancel.domain.GetAccountByIdResult;
+import cancel.domain.GetOrderByIdInfo;
+import cancel.domain.GetOrderResult;
+import cancel.domain.NotifyInfo;
+import cancel.domain.Order;
+
+public interface CancelService {
+
+    CancelOrderResult cancelOrder(CancelOrderInfo info,String loginToken,
   ↪ String loginId) throws Exception;
+
+       public boolean sendEmail(NotifyInfo notifyInfo);
+
+    public CalculateRefundResult calculateRefund(CancelOrderInfo info);
+
+    public String calculateRefund(Order order);
+
+    public ChangeOrderResult cancelFromOrder(ChangeOrderInfo info);
+
+    public ChangeOrderResult cancelFromOtherOrder(ChangeOrderInfo info);
+
+    public boolean drawbackMoney(String money,String userId);
+
+    public GetAccountByIdResult getAccount(GetAccountByIdInfo info);
+
+    public GetOrderResult getOrderByIdFromOrder(GetOrderByIdInfo info);
+
+    public GetOrderResult getOrderByIdFromOrderOther(GetOrderByIdInfo
    ↪ info);
+
+    public ChangeOrderResult updateOtherOrderStatusToCancelAsync(
    ↪ ChangeOrderInfo info) throws InterruptedException;
+
+    public Boolean drawBackMoneyForOrderCancelAsync(String money, String
    ↪ userId,String orderId, String loginToken) throws
    ↪ InterruptedException;
+
+}
```

```
diff --git a/ts-cancel-service/src/main/java/cancel/service/
    ↪ CancelServiceImpl.java b/ts-cancel-service/src/main/java/cancel/
    ↪ service/CancelServiceImpl.java
index ff8ab2b..7b19bf8 100644
--- a/ts-cancel-service/src/main/java/cancel/service/CancelServiceImpl.
    ↪ java
+++ b/ts-cancel-service/src/main/java/cancel/service/CancelServiceImpl.
    ↪ java
@@ -1,401 +1,284 @@
-package cancel.service;
-
-import cancel.async.AsyncTask;
-import cancel.domain.*;
-import org.springframework.beans.factory.annotation.Autowired;
-import org.springframework.stereotype.Service;
-import org.springframework.web.client.RestTemplate;
-import java.text.DecimalFormat;
-import java.util.Calendar;
-import java.util.Date;
-
-@Service
-public class CancelServiceImpl implements CancelService{
-
-    @Autowired
-    private RestTemplate restTemplate;
-
-    @Autowired
-    private AsyncTask asyncTask;
```

```
—
—     @Override
—     public CancelOrderResult cancelOrder(CancelOrderInfo info,String
   ↪ loginToken,String loginId) throws Exception{
—         GetOrderByIdInfo getFromOrderInfo = new GetOrderByIdInfo();
—         getFromOrderInfo.setOrderId(info.getOrderId());
—         GetOrderResult orderResult = getOrderByIdFromOrder(
   ↪ getFromOrderInfo);
—         if(orderResult.isStatus() == true){
—             System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣
   ↪ Order␣found␣G|H");
—             Order order = orderResult.getOrder();
—             if(order.getStatus() == OrderStatus.NOTPAID.getCode()
—                     || order.getStatus() == OrderStatus.PAID.getCode() ||
   ↪  order.getStatus() == OrderStatus.CHANGE.getCode()){
—
—                 order.setStatus(OrderStatus.CANCEL.getCode());
—                 ChangeOrderInfo changeOrderInfo = new ChangeOrderInfo();
—                 changeOrderInfo.setLoginToken(loginToken);
—                 changeOrderInfo.setOrder(order);
—
—
—
—
—                 ChangeOrderResult changeOrderResult = cancelFromOrder(
   ↪ changeOrderInfo);
—                 if(changeOrderResult.isStatus() == true){
```

```
-                       CancelOrderResult finalResult = new CancelOrderResult
   ↪ ();
-                       finalResult.setStatus(true);
-                       finalResult.setMessage("Success.");
-                       System.out.println("[Cancel␣Order␣Service][Cancel␣
   ↪ Order]␣Success.");
-                       //Draw back money
-                       String money = calculateRefund(order);
-                       boolean status = drawbackMoney(money,loginId);
-                       if(status == true){
-                           System.out.println("[Cancel␣Order␣Service][Draw␣
   ↪ Back␣Money]␣Success.");
-
-                           GetAccountByIdInfo getAccountByIdInfo = new
   ↪ GetAccountByIdInfo();
-                           getAccountByIdInfo.setAccountId(order.
   ↪ getAccountId().toString());
-                           GetAccountByIdResult result = getAccount(
   ↪ getAccountByIdInfo);
-                           if(result.isStatus() == false){
-                               return null;
-                           }
-
-                           NotifyInfo notifyInfo = new NotifyInfo();
-                           notifyInfo.setDate(new Date().toString());
-
-
-                           notifyInfo.setEmail(result.getAccount().getEmail
↪ ());
-                           notifyInfo.setStartingPlace(order.getFrom());
-                           notifyInfo.setEndPlace(order.getTo());
-                           notifyInfo.setUsername(result.getAccount().
↪ getName());
-                           notifyInfo.setSeatNumber(order.getSeatNumber());
-                           notifyInfo.setOrderNumber(order.getId().toString
↪ ());
-                           notifyInfo.setPrice(order.getPrice());
-                           notifyInfo.setSeatClass(SeatClass.getNameByCode(
↪ order.getSeatClass()));
-                           notifyInfo.setStartingTime(order.getTravelTime().
↪ toString());
-
-                           sendEmail(notifyInfo);
-
-                       }else{
-                           System.out.println("[Cancel␣Order␣Service][Draw␣
↪ Back␣Money]␣Fail.");
-                       }
-
-
-
-                       return finalResult;
-                   }else{
-                       CancelOrderResult finalResult = new CancelOrderResult
↪ ();
```

```
-                       finalResult.setStatus(false);
-                       finalResult.setMessage(changeOrderResult.getMessage()
  ↪ );
-                       System.out.println("[Cancel␣Order␣Service][Cancel␣
  ↪ Order]␣Fail.Reason:" + changeOrderResult.getMessage());
-                       return finalResult;
-                   }
-
-               }else{
-                   CancelOrderResult result = new CancelOrderResult();
-                   result.setStatus(false);
-                   result.setMessage("Order␣Status␣Cancel␣Not␣Permitted");
-                   System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣
  ↪ Order␣Status␣Not␣Permitted.");
-                   return result;
-               }
-           }else{
-               GetOrderByIdInfo getFromOtherOrderInfo = new GetOrderByIdInfo
  ↪ ();
-               getFromOtherOrderInfo.setOrderId(info.getOrderId());
-               GetOrderResult orderOtherResult = getOrderByIdFromOrderOther(
  ↪ getFromOtherOrderInfo);
-               if(orderOtherResult.isStatus() == true){
-                   System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣
  ↪ Order␣found␣Z|K|Other");
-
-                   Order order = orderOtherResult.getOrder();
-                   if(order.getStatus() == OrderStatus.NOTPAID.getCode()
-                               || order.getStatus() == OrderStatus.PAID.getCode
   ↪ () || order.getStatus() == OrderStatus.CHANGE.getCode()){
-
-                         System.out.println("[Cancel␣Order␣Service][Cancel␣
   ↪ Order]␣Order␣status␣ok");
-
-                         order.setStatus(OrderStatus.CANCEL.getCode());
-                         ChangeOrderInfo changeOrderInfo = new ChangeOrderInfo
   ↪ ();
-                         changeOrderInfo.setLoginToken(loginToken);
-                         changeOrderInfo.setOrder(order);
-                         ChangeOrderResult changeOrderResult =
   ↪ cancelFromOtherOrder(changeOrderInfo);
-
-
-                         if(changeOrderResult.isStatus() == true){
-                              CancelOrderResult finalResult = new
   ↪ CancelOrderResult();
-                              finalResult.setStatus(true);
-                              finalResult.setMessage("Success.");
-                              System.out.println("[Cancel␣Order␣Service][Cancel
   ↪ ␣Order]␣Success.");
-                              //Draw back money
-                              String money = calculateRefund(order);
-                              boolean status = drawbackMoney(money,loginId);
-                              if(status == true){
-                                   System.out.println("[Cancel␣Order␣Service][
   ↪ Draw␣Back␣Money]␣Success.");
—                           }else{
—                               System.out.println("[Cancel␣Order␣Service][
   ↪ Draw␣Back␣Money]␣Fail.");
—                           }
—                           return finalResult;
—                       }else{
—                           CancelOrderResult finalResult = new
   ↪ CancelOrderResult();
—                           finalResult.setStatus(false);
—                           finalResult.setMessage(changeOrderResult.
   ↪ getMessage());
—                           System.out.println("[Cancel␣Order␣Service][Cancel
   ↪ ␣Order]␣Fail.Reason:" + changeOrderResult.getMessage());
—                           return finalResult;
—                       }
—                   }else{
—                       CancelOrderResult result = new CancelOrderResult();
—                       result.setStatus(false);
—                       result.setMessage("Order␣Status␣Cancel␣Not␣Permitted"
   ↪ );
—                       System.out.println("[Cancel␣Order␣Service][Cancel␣
   ↪ Order]␣Order␣Status␣Not␣Permitted.");
—                       return result;
—                   }
—               }else{
—                   CancelOrderResult result = new CancelOrderResult();
—                   result.setStatus(false);
—                   result.setMessage("Order␣Not␣Found");
```

```
-                System.out.println("[Cancel␣Order␣Service][Cancel␣Order]␣
   ↪ Order␣Not␣Found.");
-                return result;
-            }
-        }
-    }
-
-    public boolean sendEmail(NotifyInfo notifyInfo){
-        System.out.println("[Cancel␣Order␣Service][Send␣Email]");
-        boolean result = restTemplate.postForObject(
-                "http://ts-notification-service:17853/notification/
   ↪ order_cancel_success",
-                notifyInfo,
-                Boolean.class
-        );
-        return result;
-    }
-
-
-    public CalculateRefundResult calculateRefund(CancelOrderInfo info){
-        GetOrderByIdInfo getFromOrderInfo = new GetOrderByIdInfo();
-        getFromOrderInfo.setOrderId(info.getOrderId());
-        GetOrderResult orderResult = getOrderByIdFromOrder(
   ↪ getFromOrderInfo);
-        if(orderResult.isStatus() == true){
-            Order order = orderResult.getOrder();
-            if(order.getStatus() == OrderStatus.NOTPAID.getCode()
-                    || order.getStatus() == OrderStatus.PAID.getCode()){
```

```
—                   if(order.getStatus() == OrderStatus.NOTPAID.getCode()){
—                       CalculateRefundResult result = new
    ↪ CalculateRefundResult();
—                       result.setStatus(true);
—                       result.setMessage("Success");
—                       result.setRefund("0");
—                       System.out.println("[Cancel␣Order][Refund␣Price]␣From
    ↪ ␣Order␣Service.Not␣Paid.");
—                       return result;
—                   }else{
—                       CalculateRefundResult result = new
    ↪ CalculateRefundResult();
—                       result.setStatus(true);
—                       result.setMessage("Success");
—                       result.setRefund(calculateRefund(order));
—                       System.out.println("[Cancel␣Order][Refund␣Price]␣From
    ↪ ␣Order␣Service.Paid.");
—                       return result;
—                   }
—               }else{
—                   CalculateRefundResult result = new CalculateRefundResult
    ↪ ();
—                   result.setStatus(false);
—                   result.setMessage("Order␣Status␣Cancel␣Not␣Permitted");
—                   result.setRefund("error");
—                   System.out.println("[Cancel␣Order][Refund␣Price]␣Order.␣
    ↪ Cancel␣Not␣Permitted.");
—
```

```
-                   return result;
-               }
-           }else{
-               GetOrderByIdInfo getFromOtherOrderInfo = new GetOrderByIdInfo
   ↪ ();
-               getFromOtherOrderInfo.setOrderId(info.getOrderId());
-               GetOrderResult orderOtherResult = getOrderByIdFromOrderOther(
   ↪ getFromOtherOrderInfo);
-               if(orderOtherResult.isStatus() == true){
-                   Order order = orderOtherResult.getOrder();
-                   if(order.getStatus() == OrderStatus.NOTPAID.getCode()
-                           || order.getStatus() == OrderStatus.PAID.getCode
   ↪ ()){
-                       if(order.getStatus() == OrderStatus.NOTPAID.getCode()
   ↪ ){
-                           CalculateRefundResult result = new
   ↪ CalculateRefundResult();
-                           result.setStatus(true);
-                           result.setMessage("Success");
-                           result.setRefund("0");
-                           System.out.println("[Cancel␣Order][Refund␣Price]␣
   ↪ From␣Order␣Other␣Service.Not␣Paid.");
-                           return result;
-                       }else{
-                           CalculateRefundResult result = new
   ↪ CalculateRefundResult();
-                           result.setStatus(true);
-                           result.setMessage("Success");
—                        result.setRefund(calculateRefund(order));
—                        System.out.println("[Cancel␣Order][Refund␣Price]␣
   ↪ From␣Order␣Other␣Service.Paid.");
—                        return result;
—                    }
—                }else{
—                    CalculateRefundResult result = new
   ↪ CalculateRefundResult();
—                    result.setStatus(false);
—                    result.setMessage("Order␣Status␣Cancel␣Not␣Permitted"
   ↪ );
—                    result.setRefund("error");
—                    System.out.println("[Cancel␣Order][Refund␣Price]␣
   ↪ Order␣Other.␣Cancel␣Not␣Permitted.");
—                    return result;
—                }
—            }else{
—                CalculateRefundResult result = new CalculateRefundResult
   ↪ ();
—                result.setStatus(false);
—                result.setMessage("Order␣Not␣Found");
—                result.setRefund("error");
—                System.out.println("[Cancel␣Order][Refund␣Price]␣Order␣
   ↪ not␣found.");
—                return result;
—            }
—        }
—    }
```

```
-
-    private String calculateRefund(Order order){
-        if(order.getStatus() == OrderStatus.NOTPAID.getCode()){
-            return "0.00";
-        }
-        System.out.println("[Cancel␣Order]␣Order␣Travel␣Date:" + order.
↪ getTravelDate().toString());
-        Date nowDate = new Date();
-        Calendar cal = Calendar.getInstance();
-        cal.setTime(order.getTravelDate());
-        int year = cal.get(Calendar.YEAR);
-        int month = cal.get(Calendar.MONTH);
-        int day = cal.get(Calendar.DAY_OF_MONTH);
-        Calendar cal2 = Calendar.getInstance();
-        cal2.setTime(order.getTravelTime());
-        int hour = cal2.get(Calendar.HOUR);
-        int minute = cal2.get(Calendar.MINUTE);
-        int second = cal2.get(Calendar.SECOND);
-        Date startTime = new Date(year,
-                                  month,
-                                  day,
-                                  hour,
-                                  minute,
-                                  second);
-        System.out.println("[Cancel␣Order]␣nowDate␣␣:" + nowDate.toString
↪ ());
-        System.out.println("[Cancel␣Order]␣startTime:" + startTime.
↪ toString());
```

```
-           if(nowDate.after(startTime)){
-               System.out.println("[Cancel␣Order]␣Ticket␣expire␣refund␣0");
-               return "0";
-           }else{
-               double totalPrice = Double.parseDouble(order.getPrice());
-               double price = totalPrice * 0.8;
-               DecimalFormat priceFormat = new java.text.DecimalFormat("0.00
    ↪ ");
-               String str = priceFormat.format(price);
-               System.out.println("[Cancel␣Order]calculate␣refund␣-␣" + str)
    ↪ ;
-               return str;
-           }
-       }
-
-
-       private ChangeOrderResult cancelFromOrder(ChangeOrderInfo info){
-           System.out.println("[Cancel␣Order␣Service][Change␣Order␣Status]␣
    ↪ Changing....");
-           ChangeOrderResult result = restTemplate.postForObject("http://ts-
    ↪ order-service:12031/order/update",info,ChangeOrderResult.class);
-           return result;
-       }
-
-       private ChangeOrderResult cancelFromOtherOrder(ChangeOrderInfo info){
-           System.out.println("[Cancel␣Order␣Service][Change␣Order␣Status]␣
    ↪ Changing....");
```

```
-         ChangeOrderResult result = restTemplate.postForObject("http://ts-
↪ order-other-service:12032/orderOther/update",info,ChangeOrderResult.
↪ class);
-         return result;
-     }
-
-     public boolean drawbackMoney(String money,String userId){
-         System.out.println("[Cancel␣Order␣Service][Draw␣Back␣Money]␣Draw␣
↪ back␣money...");
-         DrawBackInfo info = new DrawBackInfo();
-         info.setMoney(money);
-         info.setUserId(userId);
-         String result = restTemplate.postForObject("http://ts-inside-
↪ payment-service:18673/inside_payment/drawBack",info,String.class);
-         if(result.equals("true")){
-             return true;
-         }else{
-             return false;
-         }
-     }
-
-     public GetAccountByIdResult getAccount(GetAccountByIdInfo info){
-         System.out.println("[Cancel␣Order␣Service][Get␣By␣Id]");
-         GetAccountByIdResult result = restTemplate.postForObject(
-                 "http://ts-sso-service:12349/account/findById",
-                 info,
-                 GetAccountByIdResult.class
-         );
-           return result;
-       }
-
-       private GetOrderResult getOrderByIdFromOrder(GetOrderByIdInfo info){
-           System.out.println("[Cancel␣Order␣Service][Get␣Order]␣Getting....
    ↪ ");
-           GetOrderResult cor = restTemplate.postForObject(
-                   "http://ts-order-service:12031/order/getById/"
-                   ,info,GetOrderResult.class);
-           return cor;
-       }
-
-       private GetOrderResult getOrderByIdFromOrderOther(GetOrderByIdInfo
    ↪ info){
-           System.out.println("[Cancel␣Order␣Service][Get␣Order]␣Getting....
    ↪ ");
-           GetOrderResult cor = restTemplate.postForObject(
-                   "http://ts-order-other-service:12032/orderOther/getById/"
-                   ,info,GetOrderResult.class);
-           return cor;
-       }
-
-}
+package cancel.service;
+
+import cancel.async.AsyncTask;
+import cancel.domain.*;
+import cancel.workflow.CancelWorkflow;
```

```
+import io.temporal.api.common.v1.WorkflowExecution;
+import io.temporal.client.WorkflowClient;
+import io.temporal.client.WorkflowOptions;
+import io.temporal.serviceclient.WorkflowServiceStubs;
+import io.temporal.serviceclient.WorkflowServiceStubsOptions;
+
+import org.springframework.beans.factory.annotation.Autowired;
+import org.springframework.scheduling.annotation.AsyncResult;
+import org.springframework.stereotype.Service;
+import org.springframework.web.client.RestTemplate;
+import java.text.DecimalFormat;
+import java.util.Calendar;
+import java.util.Date;
+import java.util.UUID;
+import java.util.concurrent.CompletableFuture;
+import java.util.concurrent.Future;
+
+@Service
+public class CancelServiceImpl implements CancelService {
+
+       private String hostname = "134.129.91.178";
+
+       @Autowired
+       private RestTemplate restTemplate = new RestTemplate();
+
+       @Override
+       public CancelOrderResult cancelOrder(CancelOrderInfo info, String
    ↪ loginToken, String loginId) throws Exception {
```

```
+

// Initiator
+                // WorkflowServiceStubs is a gRPC stubs wrapper that talks
    ↪  to the local Docker
+                // instance of the Temporal server.
+                WorkflowServiceStubsOptions stubOptions =
    ↪ WorkflowServiceStubsOptions.newBuilder().setTarget(hostname+":7233")
+                                .build();
+                WorkflowServiceStubs service = WorkflowServiceStubs.
    ↪ newServiceStubs(stubOptions);
+                WorkflowOptions options = WorkflowOptions.newBuilder().
    ↪ setTaskQueue("default")
+                                // A WorkflowId prevents this it from
    ↪ having duplicate instances, remove it to
+                                // duplicate.
+                                .setWorkflowId("cancel-order-workflow").
    ↪ build();
+                // WorkflowClient can be used to start, signal, query,
    ↪ cancel, and terminate
+                // Workflows.
+                WorkflowClient client = WorkflowClient.newInstance(service
    ↪ );
+                // WorkflowStubs enable calls to methods as if the
    ↪ Workflow object is local, but
+                // actually perform an RPC.
+                CancelWorkflow workflow = client.newWorkflowStub(
    ↪ CancelWorkflow.class, options);
```

```
+                return workflow.cancel(info, loginToken, loginId);
+                // Asynchronous execution. This process will exit after
    ↪ making this call.
+                // CompletableFuture<CancelOrderResult> we =
+                // WorkflowClient.execute(workflow::cancel, info,
    ↪ loginToken, loginId);
+                // System.out.printf("\nTransfer of $%f from account %s to
    ↪  account %s is
+                // processing\n", amount, fromAccount, toAccount);
+                // System.out.printf("\nWorkflowID: %s RunID: %s", we.
    ↪ getWorkflowId(),
+                // we.getRunId());
+        }
+
+        @Override
+        public boolean sendEmail(NotifyInfo notifyInfo) {
+                System.out.println("[Cancel␣Order␣Service][Send␣Email]");
+                boolean result = restTemplate.postForObject(
+                                "http://"+hostname+":17853/notification/
    ↪ order_cancel_success", notifyInfo, Boolean.class);
+                return result;
+        }
+
+        @Override
+        public CalculateRefundResult calculateRefund(CancelOrderInfo info)
    ↪  {
+                GetOrderByIdInfo getFromOrderInfo = new GetOrderByIdInfo()
    ↪ ;
```

```
+                   getFromOrderInfo.setOrderId(info.getOrderId());
+                   GetOrderResult orderResult = getOrderByIdFromOrder(
    ↪ getFromOrderInfo);
+                   if (orderResult.isStatus() == true) {
+                           Order order = orderResult.getOrder();
+                           if (order.getStatus() == OrderStatus.NOTPAID.
    ↪ getCode() || order.getStatus() == OrderStatus.PAID.getCode()) {
+                                   if (order.getStatus() == OrderStatus.
    ↪ NOTPAID.getCode()) {
+                                           CalculateRefundResult result = new
    ↪  CalculateRefundResult();
+                                           result.setStatus(true);
+                                           result.setMessage("Success.
    ↪ Calculate␣Refund␣Not␣paid");
+                                           result.setRefund("0");
+                                           System.out.println("[Cancel␣Order
    ↪ ][Refund␣Price]␣From␣Order␣Service.Not␣Paid.");
+                                           return result;
+                                   } else {
+                                           CalculateRefundResult result = new
    ↪  CalculateRefundResult();
+                                           result.setStatus(true);
+                                           result.setMessage("Success.
    ↪ Calculate␣Refund");
+                                           result.setRefund(calculateRefund(
    ↪ order));
+                                           System.out.println("[Cancel␣Order
    ↪ ][Refund␣Price]␣From␣Order␣Service.Paid.");
```

```
+                                       return result;
+                               }
+                       } else {
+                               CalculateRefundResult result = new
  ↪ CalculateRefundResult();
+                               result.setStatus(false);
+                               result.setMessage("Order␣Status␣Cancel␣Not
  ↪ ␣Permitted");
+                               result.setRefund("error");
+                               System.out.println("[Cancel␣Order][Refund␣
  ↪ Price]␣Order.␣Cancel␣Not␣Permitted.");
+
+                               return result;
+                       }
+               } else {
+                       GetOrderByIdInfo getFromOtherOrderInfo = new
  ↪ GetOrderByIdInfo();
+                       getFromOtherOrderInfo.setOrderId(info.getOrderId()
  ↪ );
+                       GetOrderResult orderOtherResult =
  ↪ getOrderByIdFromOrderOther(getFromOtherOrderInfo);
+                       if (orderOtherResult.isStatus() == true) {
+                               Order order = orderOtherResult.getOrder();
+                               if (order.getStatus() == OrderStatus.
  ↪ NOTPAID.getCode()
+                                               || order.getStatus() ==
  ↪ OrderStatus.PAID.getCode()) {
```

```
+                                               if (order.getStatus() ==
    ↪ OrderStatus.NOTPAID.getCode()) {
+                                                         CalculateRefundResult
    ↪ result = new CalculateRefundResult();
+                                                         result.setStatus(true);
+                                                         result.setMessage("Success
    ↪ .Calculate␣not␣pay");
+                                                         result.setRefund("0");
+                                                         System.out.println("[
    ↪ Cancel␣Order][Refund␣Price]␣From␣Order␣Other␣Service.Not␣Paid.");
+                                                         return result;
+                                               } else {
+                                                         CalculateRefundResult
    ↪ result = new CalculateRefundResult();
+                                                         result.setStatus(true);
+                                                         result.setMessage("Success
    ↪ .Calculate␣pay");
+                                                         result.setRefund(
    ↪ calculateRefund(order));
+                                                         System.out.println("[
    ↪ Cancel␣Order][Refund␣Price]␣From␣Order␣Other␣Service.Paid.");
+                                                         return result;
+                                               }
+                                       } else {
+                                               CalculateRefundResult result = new
    ↪  CalculateRefundResult();
+                                               result.setStatus(false);
+                                               result.setMessage("Order␣Status␣
↪ Cancel␣Not␣Permitted");
+                                               result.setRefund("error");
+                                               System.out.println("[Cancel␣Order
↪ ][Refund␣Price]␣Order␣Other.␣Cancel␣Not␣Permitted.");
+                                               return result;
+                                       }
+                               } else {
+                                       CalculateRefundResult result = new
↪ CalculateRefundResult();
+                                       result.setStatus(false);
+                                       result.setMessage("Order␣Not␣Found");
+                                       result.setRefund("error");
+                                       System.out.println("[Cancel␣Order][Refund␣
↪ Price]␣Order␣not␣found.");
+                                       return result;
+                               }
+                       }
+       }
+
+       @Override
+       public String calculateRefund(Order order) {
+               if (order.getStatus() == OrderStatus.NOTPAID.getCode()) {
+                       return "0.00";
+               }
+               System.out.println("[Cancel␣Order]␣Order␣Travel␣Date:" +
↪ order.getTravelDate().toString());
+               Date nowDate = new Date();
+                 Calendar cal = Calendar.getInstance();
+                 cal.setTime(order.getTravelDate());
+                 int year = cal.get(Calendar.YEAR);
+                 int month = cal.get(Calendar.MONTH);
+                 int day = cal.get(Calendar.DAY_OF_MONTH);
+                 Calendar cal2 = Calendar.getInstance();
+                 cal2.setTime(order.getTravelTime());
+                 int hour = cal2.get(Calendar.HOUR);
+                 int minute = cal2.get(Calendar.MINUTE);
+                 int second = cal2.get(Calendar.SECOND);
+                 Date startTime = new Date(year, month, day, hour, minute,
    ↪ second);
+                 System.out.println("[Cancel␣Order]␣nowDate␣␣:" + nowDate.
    ↪ toString());
+                 System.out.println("[Cancel␣Order]␣startTime:" + startTime
    ↪ .toString());
+                 if (nowDate.after(startTime)) {
+                          System.out.println("[Cancel␣Order]␣Ticket␣expire␣
    ↪ refund␣0");
+                          return "0";
+                 } else {
+                          double totalPrice = Double.parseDouble(order.
    ↪ getPrice());
+                          double price = totalPrice * 0.8;
+                          DecimalFormat priceFormat = new java.text.
    ↪ DecimalFormat("0.00");
+                          String str = priceFormat.format(price);
+                               System.out.println("[Cancel␣Order]calculate␣refund
    ↪ ␣-␣" + str);
+                               return str;
+                     }
+           }
+
+           @Override
+           public ChangeOrderResult cancelFromOrder(ChangeOrderInfo info) {
+                     System.out.println("[Cancel␣Order␣Service][Change␣Order␣
    ↪ Status]␣Changing....");
+                     ChangeOrderResult result = restTemplate.postForObject("
    ↪ http://"+hostname+":12031/order/update", info,
+                                         ChangeOrderResult.class);
+                     return result;
+           }
+
+           @Override
+           public ChangeOrderResult cancelFromOtherOrder(ChangeOrderInfo info
    ↪ ) {
+                     System.out.println("[Cancel␣Order␣Service][Change␣Order␣
    ↪ Status]␣Changing....");
+                     ChangeOrderResult result = restTemplate.postForObject("
    ↪ http://"+hostname+":12032/orderOther/update",
+                                         info, ChangeOrderResult.class);
+                     return result;
+           }
+
+           @Override
+       public boolean drawbackMoney(String money, String userId) {
+               System.out.println("[Cancel␣Order␣Service][Draw␣Back␣Money
 ↪ ]␣Draw␣back␣money...");
+               DrawBackInfo info = new DrawBackInfo();
+               info.setMoney(money);
+               info.setUserId(userId);
+               String result = restTemplate.postForObject("http://"+
 ↪ hostname+":18673/inside_payment/drawBack",
+                               info, String.class);
+               if (result.equals("true")) {
+                       return true;
+               } else {
+                       return false;
+               }
+       }
+
+       @Override
+       public GetAccountByIdResult getAccount(GetAccountByIdInfo info) {
+               System.out.println("[Cancel␣Order␣Service][Get␣By␣Id]");
+               GetAccountByIdResult result = restTemplate.postForObject("
 ↪ http://"+hostname+":12349/account/findById", info,
+                               GetAccountByIdResult.class);
+               return result;
+       }
+
+       @Override
+       public GetOrderResult getOrderByIdFromOrder(GetOrderByIdInfo info)
 ↪ {
```

```
+                System.out.println("[Cancel␣Order␣Service][Get␣Order]␣
    ↪ Getting....");
+                GetOrderResult cor = restTemplate.postForObject("http://"+
    ↪ hostname+":12031/order/getById/", info,
+                                        GetOrderResult.class);
+                return cor;
+        }
+
+        @Override
+        public GetOrderResult getOrderByIdFromOrderOther(GetOrderByIdInfo
    ↪ info) {
+                System.out.println("[Cancel␣Order␣Service][Get␣Order]␣
    ↪ Getting....");
+                GetOrderResult cor = restTemplate.postForObject("http://"+
    ↪ hostname+":12032/orderOther/getById/", info,
+                                        GetOrderResult.class);
+                return cor;
+        }
+
+    public ChangeOrderResult updateOtherOrderStatusToCancelAsync(
    ↪ ChangeOrderInfo info) throws InterruptedException{
+
+         Thread.sleep(4000);
+
+         System.out.println("[Cancel␣Order␣Service][Change␣Order␣Status]")
    ↪ ;
+         ChangeOrderResult result = restTemplate.postForObject("http://"+
    ↪ hostname+":12032/orderOther/update",info,ChangeOrderResult.class);
```

```
+           return result;
+
+     }
+
+     public Boolean drawBackMoneyForOrderCancelAsync(String money, String
    ↪ userId,String orderId, String loginToken) throws
    ↪ InterruptedException {
+         //return asyncTask.drawBackMoneyForOrderCancel(money, userId,
    ↪ orderId, loginToken);
+                     /*********************** Fault Reproduction -
    ↪ Error Process Seq ************************/
+                 //double op = new Random().nextDouble();
+                 //if(op < 1.0){
+                 //     System.out.println("[Cancel Order Service] Delay
    ↪ Process Wrong Cancel Process");
+                     //Thread.sleep(8000);
+                 //} else {
+                 //     System.out.println("[Cancel Order Service] Normal
    ↪ Process Normal Cancel Process");
+                 //}
+
+
+                 //1.Search Order Info
+                 System.out.println("[Cancel␣Order␣Service][Get␣Order]␣
    ↪ Getting....");
+                 GetOrderByIdInfo getOrderInfo = new GetOrderByIdInfo();
+                 getOrderInfo.setOrderId(orderId);
+                 GetOrderResult cor = restTemplate.postForObject(
+                                "http://"+hostname+":12032/orderOther/getById/"
+                                ,getOrderInfo,GetOrderResult.class);
+                        Order order = cor.getOrder();
+                        System.out.println("[Cancel␣Order␣Service]Got␣order␣
    ↪ successfully");
+
+                        //2.Change order status to cancelling
+                        order.setStatus(OrderStatus.Canceling.getCode());
+                        ChangeOrderInfo changeOrderInfo = new ChangeOrderInfo();
+                        changeOrderInfo.setOrder(order);
+                        changeOrderInfo.setLoginToken(loginToken);
+                        ChangeOrderResult changeOrderResult = restTemplate.
    ↪ postForObject("http://"+hostname+":12032/orderOther/update",
    ↪ changeOrderInfo,ChangeOrderResult.class);
+                        if(changeOrderResult.isStatus() == false){
+                            System.out.println("[Cancel␣Order␣Service]Unexpected␣
    ↪ error");
+                        }
+                        //3.do drawback money
+                        System.out.println("[Cancel␣Order␣Service][Draw␣Back␣Money
    ↪ ]␣Draw␣back␣money...");
+                        DrawBackInfo info = new DrawBackInfo();
+                        info.setMoney(money);
+                        info.setUserId(userId);
+                        String result = restTemplate.postForObject("http://"+
    ↪ hostname+":18673/inside_payment/drawBack",info,String.class);
+                        if(result.equals("true")){
+                            return true;
+                       }else{
+                             return false;
+                       }
+                       /*********************/
+       }
+
+}
```