Extract: Informing Microservice Extraction Decisions From the Bottom-up

by

Shizuko Akamoto

BSc. Computer Science, University of British Columbia, 2021

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Extract: Informing Microservice Extraction Decisions From the Bottom-up

submitted by **Shizuko Akamoto** in partial fulfillment of the requirements for the degree of **Master of Science** in **Computer Science**.

Examining Committee:

Reid Holmes, Professor, Computer Science, UBC
Supervisor

Elisa Baniassad, Professor of Teaching, Computer Science, UBC
Supervisory Committee Member

Abstract

Microservice architectures are a popular approach for modernizing legacy monolithic systems. However, transforming an existing monolith into microservices is a complex task for developers. In practice, developers typically perform the decomposition iteratively, extracting one service at a time. Unfortunately, most existing tools do not adequately support such extraction tasks, as they aim to transform a monolith into microservices architecture at once. This paper presents Extract, a prototype tool that aligns with the developer's individual workflow for performing a single microservice extraction task. Extract empowers developers to analyze the particular classes and methods within the monolith that they intend to migrate to a new microservice. Moreover, it offers human-in-the-loop support, allowing developers to make well-informed decisions regarding the extraction process and explore alternative designs. We evaluate Extract with four case studies involving open-source monolithic applications. These studies serve to demonstrate how the approach aids developers in making practical decisions during the extraction process. Furthermore, we compare the outcomes of Extract's service designs with manually decomposed microservices for further analysis.

Lay Summary

This paper discusses the challenge of modernizing an legacy monolithic system using microservices architecture. Typically, developers break down the monolith into smaller services, one at a time. Existing tools fall short in supporting step-by-step decomposition, complicating the decomposition process. To address this, the paper introduces *Extract*, an iterative decomposition tool. Extract facilitates developers to choose components for extraction and provides informed decision-making guidance. We evaluated the tool with four case studies using open-source monolithic applications. Extract enabled the developer to make practical extraction decision and produced service designs that are comparable to manually-performed decomposition.

Preface

The work presented in this thesis was conducted in the Software Practices Laboratory at the University of British Columbia, Point Grey campus.

Shizuko Akamoto was the lead investigator, responsible for concept formation, implementation, case studies, and result analysis. Reid Holmes was the supervisory author on this work and was involved throughout all stages of the project.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I am extremely grateful for the support and encouragement from Reid Holmes throughout the duration of this work. After each meeting, I walked away with something new—whether it was a fresh perspective on the work, an unexplored idea to experiment with, or a clear direction to focus my efforts.

I could not have undertaken on this journey without the constant motivation and inspiration from my research companions, Marie Salomon and Katharine Kerr. Both of you consistently believed in my potential and offered invaluable guidance during moments of uncertainty. I am incredibly proud that, after numerous coffee walks, "anti-procrastination" research meetings, and motivational brunches and happy hours, we have all successfully reached the end of this journey together.

I also want to express my gratitude to everyone in my life who has enriched the two years I spent in academia. To my parents, thank you for being my greatest champions and for providing me with unwavering emotional and financial support. To Elisa, thank you for serving as my second reader and for showing me how to become a more effective TA through your remarkable patience with students. To the members of the SPL, I appreciate all of you for creating a fun and comfortable space to work and socialize in. To my friends from high school and undergraduate years, thank you for giving me much-needed breaks from research while also helping me stay on track by constantly asking about my progress. It feels good to finally be able to say "I am graduating."

Lastly, I want to extend a special thanks to my cat, Mochi, for accompanying me everyday through this journey. Mochi made a unique contribution to my work by occasionally taking a stroll across my keyboard.

Chapter 1

Introduction

Microservice-based architectures have emerged as a popular architectural style for modernizing systems for increased scalabiltiy, flexibility, and productivity [10]. Contrary to conventional monolithic applications, microservices architectures break down applications into distinct independently deployable "microservices", which coordinate with each other over the network. The modular nature of microservices offers numerous benefits to modern industrial systems. Notably, it enables the system to adapt to changing capacity requirements by scaling up or down specific parts of a system by deploying or removing the services [25]. Companies like Netflix [21], Uber [13], and SoundCloud [3] have released engineering reports describing the benefits of adopting the microservices architecture for their systems. However, despite the benefits, most systems do not initially adopt a fully-fledged microservices approach. Rather, microservices architecture tend to be the outcome of a long-term transition of an existing monolithic system [10].

When introducing system-wide architectural changes, developers face two options: to rewrite or to refactor. It is often tempting, in the face of technical debt, to start afresh by rewriting the legacy system. Yet, numerous industry case studies have shed light on the risks in such *big bang rewrites* —a complete overhaul of a legacy system (e.g., [34, 39, 40]). In contrast to architectural changes through rewriting, refactoring is the process of evolving an existing system from its current state to an improved state [23]. One specific refactoring pattern, coined the "Strangler Fig" by Martin Fowler, describes a method to gradually refactor legacy

monolithic systems by systematically replacing their components with microservices until the entire monolith is replaced [9]. The replacement proceeds iteratively, extracting features into new services, until the old monolith is "strangled" and is made obsolete by the extracted microservices. Contrary to a big-bang rewrite of a monolith into microservices, incremental replacement is significantly less intrusive to the live monolith [31]. SoundCloud has employed the iterative pattern in their microservices migration, where the legacy monolith and extracted microservices coexisted throughout the migration, communicating with each other [3].

There exist many previous research efforts to create tools for decomposing monoliths, but these tools tend to perform the decomposition in a single step (e.g., [7, 17, 19, 22, 24, 36, 41]). First, the tools perform static analysis to understand the interactions between existing code components. Next, heuristics are used to try to identify semantically-cohesive components that would be amenable to extraction. Finally, the tools automatically extract these components into new microservices [35]. While these tools are useful for advising microservice candidates, they essentially carry out sweeping large-scale changes in a single iteration, a risky action that developers try to avoid [25]. Consequently, the current tools do not align adequately with developers' preferred decomposition workflows, as they prefer iteratively examining services one by one [42].

The thesis of this work is to propose an alternative approach that addresses iterative decomposition for developers, focusing on a bottom-up strategy to enhance the decomposition process. We implement this approach as a web-based prototype called Extract. Extract helps developers identify and extract a specific feature from a monolithic Java application into a single microservice. To begin, the developer makes an initial selection of elements (i.e., methods and classes) contained in their monolith that they would like to extract into a service. Extract then statically constructs a call graph relevant to the selected code to help the developer reason about what the dependencies are between the selected elements and the rest of the system. The call graph is visually presented with supplementary summary and ease-of-extraction heuristics. Informed by the summary and heuristics, the developer makes one of three "extraction decisions" for each of the methods and classes relevant to the extracted service. Extraction decisions represent a developer's intention for a software element during the extraction process and directly translate

into corresponding actions. Extract dynamically updates the visualization as the developer incrementally refines their decisions, until a desired microservice design is achieved.

We evaluate the prototype with four case studies using open-source monolithic applications and demonstrate how Extract can guide developers through making actionable extraction decisions. Additionally, we compare the resulting designs with corresponding services from manual decomposition. We aim to determine the efficacy and shortcomings of our approach in producing designs that are comparable to manually derived ones.

This thesis makes the following contributions:

1. A bottom-up, iterative approach to monolith decomposition, accompanied by a prototype tool implementation, called Extract, that applies this approach.

2. Four case studies using the prototype that demonstrates its capability in deriving actionable extraction decisions.

3. A report on the comparison tool-supported microservice designs to manually derived correspondence.

Chapter 2

Related Work

Although research in bottom-up, iterative microservice extraction is scarce, the intent and mechanism of our approach overlaps with many existing research efforts. We detail the prior research here, along with the set of design goals we derived by leveraging and extending on prior research.

2.1 Automated Monolith Decomposition

Several projects have examined automating the decomposition of a legacy monolith into a set of microservices (e.g., [7, 17, 19, 22, 24, 36, 41]). Ponce et al. performed a review on the different decomposition techniques that have been proposed in the literature and identified three broad categories: model-driven, static analysis-based, and dynamic analysis-based [29]. Model-driven approaches use design elements like domain entities and data flow diagrams as input, which has the benefit of being technology-agnostic, but important elements like system performance tend to be neglected. Static analysis-based techniques use source code as input which simplifies the identification of code dependencies, although they are often language specific and may encounter challenges due to the lack of domain knowledge. Finally, dynamic analysis-based techniques offer the advantage of incorporating runtime behaviors of live monolithic systems, but as with any other dynamic analysis-based tools, the information they capture is incomplete, limited to only those exercised during the analysis. Among the reviewed approaches, static

analysis-based techniques were predominant, comprising 45% of the total. These approaches encompassed various forms of source code analysis, including hybrid methods that integrated static analysis with other techniques. Most static analysis-based approaches built a graph from the source code input, to which they applied a clustering approach. We focus on two decomposition tools which capture the essence of each category of monolith decomposition.

Mazlami et al. proposed a semi-automated, formal approach for extracting microservices candidates from monolithic systems [22]. They investigated two different approaches for identifying semantically coupled classes within source code. The first approach utilized static analysis to graph the syntactic similarity between classes, to which they applied Kruskal's Minimum Spanning Tree (MST) algorithm to cluster the classes into distinct microservices. In the second approach, a model-driven method was employed, where co-change information was gathered from the commit history of the monolith. They then clustered classes by categorizing classes modified together as highly similar. Of these two approaches, static analysis-based approach proved to be the most performant, based on their quality metrics of team size reduction and average domain redundancy. However, the authors acknowledge as a limitation in their approaches that the lowest computational unit is a class. Using functions or methods as the finest level of granularity could improve the precision of the extraction task.

Nakazawa et al. later performed a reproductive study of [22] and identified that the resulting microservices generated a large amount of inter-service communication because MST clustering algorithm did not consider the degree of dependencies between the clusters [24]. Hence, they proposed an alternative strategy to identify similar classes using dynamic analysis, which involved generating calling-context trees (CCTs) for the methods profiled in a sample run of the application. The CCT-based clustering produced microservices that communicated less frequently, but the individual service sizes tended to be larger.

We propose an approach that uses static analysis similar to Mezlami et al [22], but focuses on a single microservice extraction rather than whole-system decomposition. The resulting microservice design also takes into account the frequency of inter-service communication, as the tool provides developers with dependency information gathered through static call chain analysis. This enables them to eval-

uate the tradeoff between service size and runtime performance when devising a service design.

Design Goals I

DG0 *Encourage iterative extraction* by targeting one service at a time.

DG1 Provide support to evaluate the *tradeoff of service size versus communication frequency*.

2.2 Feature-Oriented Dependency Analysis

The idea of isolating a particular component from an existing system extends beyond microservices and monolith decomposition. Dependency analysis of source code elements is a common task supported by many tools that assist in defining a feature boundary in code (e.g., [32],[16],[5],[33],[6]).

Robillard and Murphy introduced the "concern" abstraction, where a concern is any conceptual unit of source code, such as a software feature [32]. The authors implemented the Feature Exploration and Analysis tool (FEAT) to represent source code as a concern graph, and enabled developers to reason about concern-level dependencies. FEAT enabled developers to manage scattered concerns by leveraging methods, fields, and types as the fundamental units of computation. Similar to FEAT, Extract allows developers to define a feature at the granularity of individual methods, and to explore how the feature intended for extraction interacts with the rest of the monolith. However, unlike FEAT which is designed primarily for broad program comprehension tasks, Extract offers heuristic advice to enable developers to make actionable decisions specific to their microservice extraction tasks.

Holmes and Walker constructed Gilligan to help developers perform programmatic reuse tasks [16]. These tasks were defined as the pragmatic reuse of software features from one system within a new system. The tool supports developers in tracing static code dependencies to determine how much of the system would need to be reused to isolate the desired feature. Gilligan then automatically extracts the identified feature for integration into the target system. Extract draws inspiration from Gilligan's approach by identifying feature boundaries via source code depen-

dency analysis and documenting the developer's decisions within the context of their task. But Gilligan itself would be limiting for the task of monolith decomposition as the approach is tailored for code reuse. For example, Gilligan only reasons about outgoing dependencies in its views, omitting incoming dependencies, to emphasize how the reused code relies on the system. We adapt and expand the Gilligan's approach for monolith decomposition by taking into account bidirectional dependencies and introducing decisions specific to decomposition tasks.

Design Goals II

DG2 *Provide flexibility for defining a feature* by allowing to make decisions at method granularity.

DG3 *Support analysis of both incoming and outgoing dependencies* to reason about how the extracted feature is used by the remaining monolith and vice versa.

2.3 Support for Decision Making

While the concept of fully-automated microservice extraction may seem appealing, there are arguments against tools replacing the decision-making role traditionally performed by developers [4]. Instead, tools should aid and enhance the decision-making processes of the people who build and maintain software systems.

Automation has been widely applied to the field of test case generation, alleviating developers of the need for manually creating test cases in the hopes of identifying software bugs (e.g., [2],[30],[37]). Developers instead apply tools to automatically generate thousands of test cases for them. However, the generated test cases often lack readability, hindering developers from understanding the specific aspects being tested [11]. Recognizing this challenge, Brandt and Zaidman proposes a developer-centric test exploration tool that allow developers to study generated tests and decide whether to incorporate them into manually written test suite [1]. For each test case, the tool presents the applied mutation and the corresponding change in instructional coverage. With these information, developers make informed decisions to either add or ignore any given generated test case.

Gilligan incorporates decision-making support in its tool suite for pragmatic reuse [16]. For each software structural element, Gilligan supports six "triage decisions" that capture the developer's intent of reuse. For example, by tagging an element as "Accept" or "Reject", the developer can specify that the element should or should not be reused. Already-made decisions are visually recorded within the tool, providing developers with clear feedback on the progress of their reuse task. Similarly, Gilligan lists any errors and remaining elements awaiting triage, providing an overview of the remaining work.

The decision-making support tools exhibit common features that make them useful for developers. Drawing inspiration from these, we derive the following additional design goals to facilitate developers in making informed decisions about microservice extraction.

Design Goals III

DG4 *Enable developers to make concrete, actionable decisions* by offering a well-defined set of task-specific questions.

DG5 *Make evident the consequences* of each decision made by providing a visual overview of task progress and obstacles remaining to be considered.

2.4 Visualizing Structural Dependencies

As systems scale, the dependencies within them become increasingly intricate, posing a significant challenge for visualizations [28]. Attempting to visualize such complexity naively could result in information overload [15].

Graph-based visualizations are the most predominant method for representing software systems [38]. Its advantage lies in the ability to directly map software elements and relationships to a visual representation. Despite their intuitiveness, graphs often lack clarity, particularly when visualizing larger, more interconnected systems. One way to reduce cognitive load in large graphs is to enforce a hierarchical ordering of the nodes. Holmes et al. utilize a tree-based dependency view in Gilligan to structure the nodes based on containment, i.e., by organizing them

according to packages and classes [16]. Falke et al. present an alternative hierarchical organization, involving dominance analysis, where at each level all children of a dominator are merged into the dominator [8].

Metaphor-based visualizations offers an alternative approach that represents software elements as recognizable real-life objects, such as buildings and roads [38]. For example, SArF map adopts the city metaphor, where each class is depicted as a building, each software feature as a block, and the relations between classes and features as the streets connecting them [18]. ExploreViz uses a similar city metaphor to visualize the runtime behaviour of microservices and the interactions between them [19]. By associating software elements with familiar physical structures, a dependency graph can become more intuitive. This is advantageous for general program comprehension and exploration tasks; however, metaphor-based visualizations are less popular for tasks involving architectural analysis and decision-making [38]. Fitting the entire system in such metaphor suffers from the same drawback as a naive graph-based visualization: information overload.

Software dependencies can also be visualized using a matrix, for example in tabular form [38]. Software elements are arranged in rows and columns, with cells denoting the relationship between the intersecting elements. Matrix-based visualizations are especially suitable for presenting dense dependency graphs or presenting supplementary information alongside the dependencies [12]. Laval et al. enhanced the cells of dependency matrices by incorporating information about the type and proportion of dependence [20]. Through case studies, developers found the enriched dependency matrix useful for identifying and understanding circular dependencies within a system. However, the study also revealed that some developers found such matrix representation less intuitive compared to alternative methods such as those based on graphs.

Considering both conciseness and intuitiveness, Extract incorporates both tree-based and matrix-based visualization in its dependency view. Similar to [20], it leverages the conciseness of a matrix to display summary and heuristic for each dependency. Furthermore, instead of having single-level column and row headers, rows and columns uses trees to hierarchically arrange the software elements based on package, class, and method containment.

Design Goals IV

DG6 *Concisely represent the dependencies* and their associated summary and heuristic information.

DG7 *Provide intuitive navigation of software elements* by hierarchically arranging them based on package, class, method containment.

Chapter 3

Microservice Extraction Process

In addition to the design goals derived from related work, we analyze a manual microservice extraction task to define a model for the extraction process. Given its recurrent appearance in case studies within multiple research literature (e.g., [14, 17]), we consider AcmeAir[1] for this manual extraction task. AcmeAir is a monolithic Java application for airline management, and we extract its booking feature as a new `Booking` microservice. The manual extraction simulates developer's approach to a single service extraction task, illustrating the necessary information for identify and extract a functionally cohesive set of code.

3.1 The Extraction Task

To extract `BookingService`, the developer needs to initially pinpoint relevant functionality in the source code. The AcmeAir codebase follows a layered three-tier architecture with a top-level REST API tier, mid-level business logic tier, and bottom database tier. By inspecting the classes in the project view, the developer identifies a group of classes potentially relevant to the booking feature, and realizes that this feature spans across multiple tiers of the application.

To carve out the feature, she begins by examining the entry point class, `BookingsREST`, and proceeds to explore its associated dependencies for other relevant classes. It becomes evident that `BookingsREST` delegates its tasks to the `BookingService`

[1] https://github.com/blueperf/acmeair-monolithic-java

interface, which, in turn, is implemented by the `BookingServiceImpl` class. Since `BookingService` solely depends on its implementer, the developer then investigates the dependencies of `BookingServiceImpl`. `BookingServiceImpl` is a database-tier class, responsible for initiating and managing communication with the underlying MongoDB database required for executing each operation provided by `BookingsREST`. To carry out its tasks, it relies on several pieces: the `ConnectionManager` class for accessing the "booking" database, the `KeyGenerator` for creating unique IDs for new bookings, and an external serialization library called `org.bson`.

Unlike the previous classes, where determining their inclusion in `BookingService` was relatively straightforward, the developer requires additional analysis to decide the appropriate course of action for `ConnectionManager`, `KeyGenerator`, and `org.bson`. Now, she needs to trace back the incoming dependencies of these classes, i.e. which other classes in AcmeAir rely on them. Upon investigation, she realizes that numerous other features in AcmeAir also make use of the methods provided by these classes. Removing them entirely from the application would result in breaking many such dependencies, and result to many inter-service REST API calls. Instead, the developer opts to incorporate the dependencies into `BookingService` by duplicating `ConnectionManager` and `KeyGenerator` classes, and adding an external library dependency for `org.bson`. The developer justifies this approach as reasonable, as each of the duplicated classes is self-contained and does not introduce extensive dependencies of its own. Furthermore, each of these self-contained set of code is a common library shared by multiple features of the system: a candidate for subsequent extraction.

Finally, the developer aims to identify the incoming dependencies of all the existing classes within `BookingService`. This analysis is crucial for determining which methods should be exposed as external APIs for the remaining AcmeAir monolith. By meticulously tracing the incoming dependencies of all 19 methods, she identifies that `BookingService.dropBookings` and `BookingService.getServiceType` receive calls from the monolith. Based on this, the developer concludes that she must externally expose the two aforementioned methods as part of `BookingService`'s API and convert method calls to them into microservice API calls.

3.2 Analysis of the Task

From the manual extraction task, we derive a model of the extraction process to which we base our approach:

1. The developer identifies an initial set of classes and methods that could be relevant to the extracted feature.

2. For each of the classes/methods within the initial extraction candidate, the developer follows its outgoing dependencies to determine the extent of extraction. For each software element it depends on, the developer decides whether to include, exclude, or duplicate the element in the new service. Throughout this process, the initial service design undergoes iterative updates as necessary.

3. The developer investigates each of the classes/methods in the updated service design (see 2), determining its incoming dependencies (i.e., what elements in the monolith use the class/method). By reviewing each dependency, the developer makes a decision whether to retain the current service design or eliminate the class/method from the service.

4. The developer iterates on steps 2 and 3 until a satisfactory result is achieved, considering the level of dependencies between the monolith and the service. In the rest of this thesis, we refer to these dependencies as *cross-boundary dependencies*.

5. Upon reaching a satisfactory service design, the developer evaluates each cross-boundary dependency and identifies which elements require refactoring to operate across the network.

6. The developer carries out the identified refactoring, thereby executing the service extraction.

3.3 Navigating and Evaluating Dependencies

The developer requires knowledge of both *outgoing* and *incoming* dependencies in order to accurately make decision regarding inclusion or exclusion of a soft-

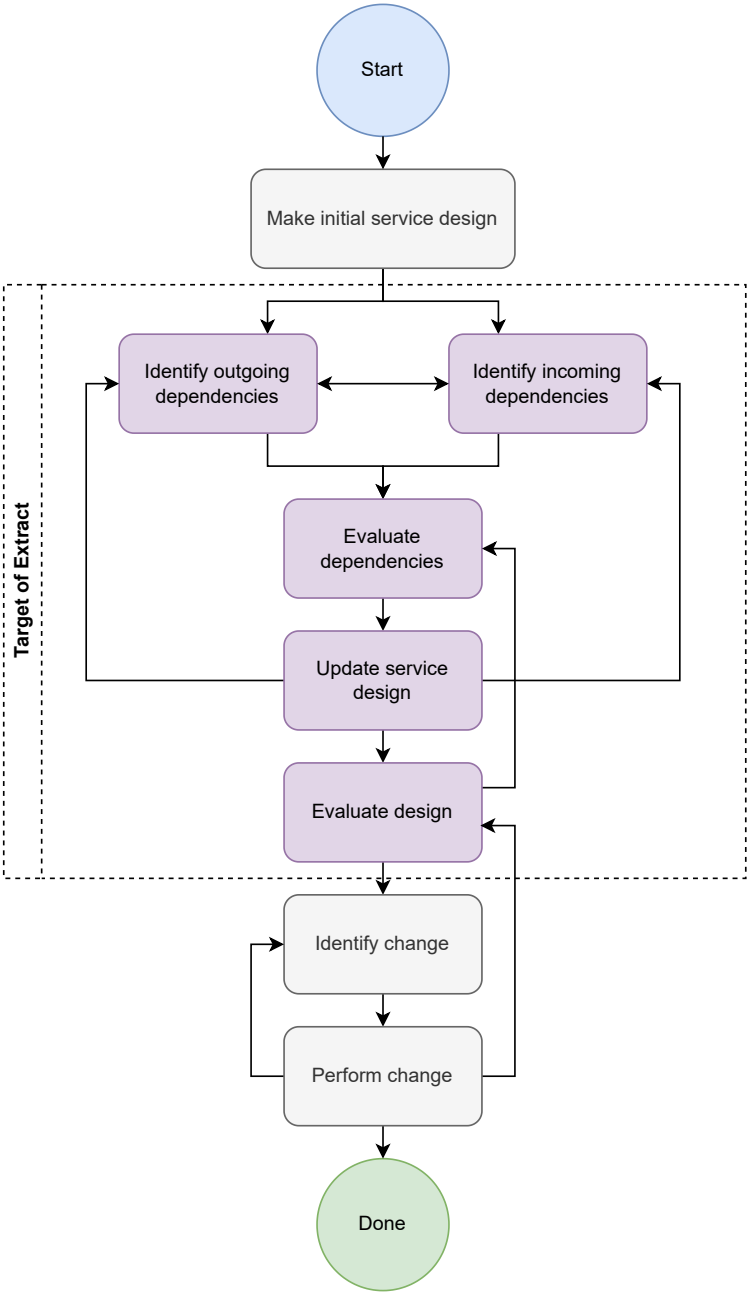

Figure 3.1: A model for an extraction process. Our approach specifically targets the steps outlined by the dotted box.

ware element from the microservice design. Together, the outgoing and incoming dependencies reflect the degree of coupling the element exhibit towards the monolith and the service. For example, elements having high number of incoming and outgoing dependencies to the monolith are tightly coupled to the monolith, and thus should be excluded from the microservice. Additionally, each identified dependency needs to be classified as either *local* or *crossing* the extraction boundary. Local dependencies exist within the bounds of the monolith or the service, whereas cross-boundary dependencies exist in-between the monolith and the service. The developer's primary objective is to minimize cross-boundary dependencies that lead to unnecessary over-the-network calls and to maximize local dependencies that increase cohesion.

A dependency of particular interest is the *circular* dependency, where two or more software elements must directly or indirectly depend on each other to function properly. In the context of microservices, a circular dependency is a cyclic chain of calls among microservices (e.g., Service `A` calls Service `B`, `B` calls `C`, and `C` calls back to `A`). This kind of dependency is considered an anti-pattern for microservices, as it hinders the independent evolution of participating services [27]. Although circular dependencies between services are typically undesirable in a finalized microservices architecture, they might be intentionally introduced during an iterative extraction process. Thus, another goal when devising an extraction design is to avoid unnecessary circular dependencies (rather than completely eliminate them) between the extracted service and the remaining monolith.

3.4 Actionable Extraction Decisions

Based on the nature of dependencies identified, the developer must make decision for each involved software elements that align with the two extraction goals mentioned earlier. We establish five *actionable extraction decisions* that represent the developer's intentions for a software element during the extraction process. Each decision focuses on one software element at a time, making the extraction design specific and, as a result, "actionable".

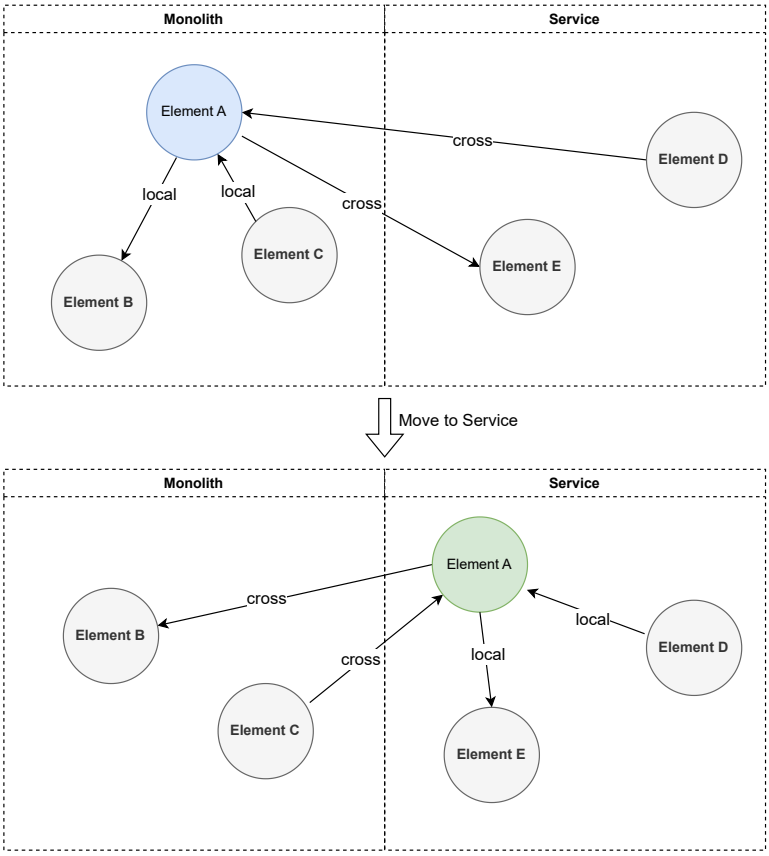

Figure 3.2: Movement to elements and dependencies in a *move to service* decision. Dependencies local to the monolith become cross-boundary, and cross-boundary dependencies become local to the service.

3.4.1 Move to service (MS)

The software element exists in the monolith in the service design, and the developer determines that it should be part of the extracted service instead. By moving the element to the service, all dependencies from/to the service become local, while all dependencies from/to the monolith become cross-boundary. (See Figure 3.2)

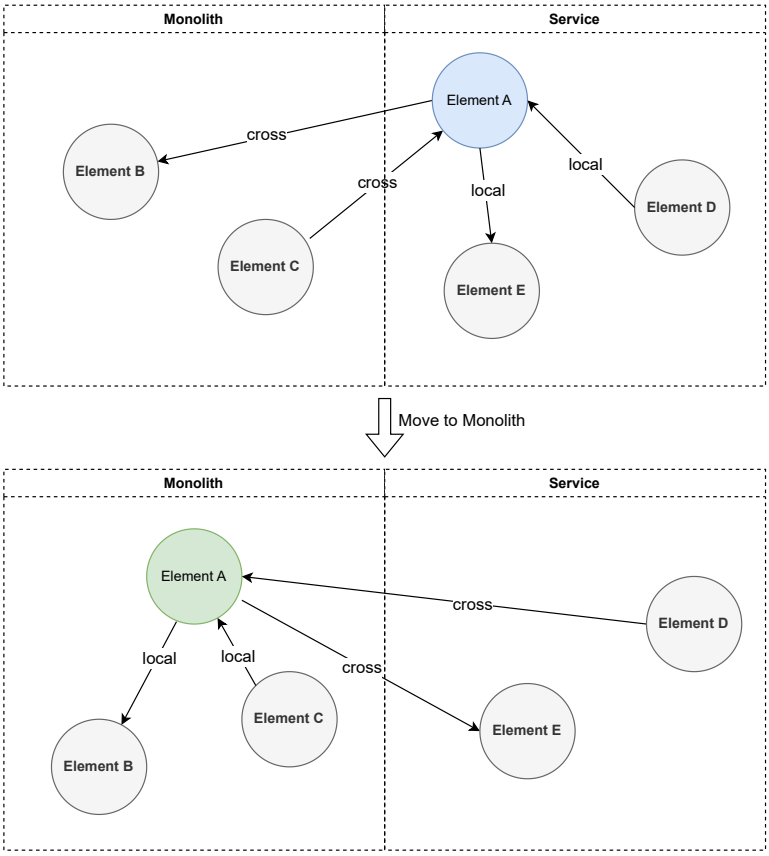

Figure 3.3: Movement to elements and dependencies in a *move to monolith* decision. Cross-boundary dependencies become local to the monolith and dependencies local to the service become cross-boundary.

3.4.2 Move to monolith (MM)

The element exists in the service in the current service design, and the developer decides that it should be part of the monolith instead. By moving the element to the monolith, all dependencies from/to the monolith become local, and all dependencies from/to the service become cross-boundary. (See Figure 3.3)

3.4.3 Confirm in service (CS)

Confirm in service decision represents that the developer is confident that an element currently in the service should indeed remain in the service, based on the analyzed dependencies. This decision does not result in any alterations to the service design or its dependencies.

3.4.4 Confirm in monolith (CM)

If the developer concludes that there is substantial evidence supporting the inclusion of an element in the monolith, then the element is tagged as confirmed in monolith.

3.4.5 Copy to service (C)

Copy to service decision is used when an element currently resides in the monolith, and the developer decides to replicate the element for utilization in the service. By copying the element, any dependencies to/from the service become local to the service, while dependencies from the copy to the monolith become cross-boundary. Note that unlike a *move* decision, dependencies *from* the monolith remain local since the element is not removed from the monolith. (See Figure 3.4)

3.5 Granularity of the Decisions

The developer requires making decisions at different levels of granularity depending on the software element at hand. We identify three types of software code:

3.5.1 Source code

In most extraction tasks, the main target of analysis is the source code, where the developer aims to isolate a specific feature from the codebase. Source code can be organized in different ways, such as by packages, by features, by layers, or by responsibilities, and the chosen structure determines whether features are closely grouped or scattered throughout the codebase [26]. This necessitates making decisions at different levels of granularity, including at package, class, method, and field-level.

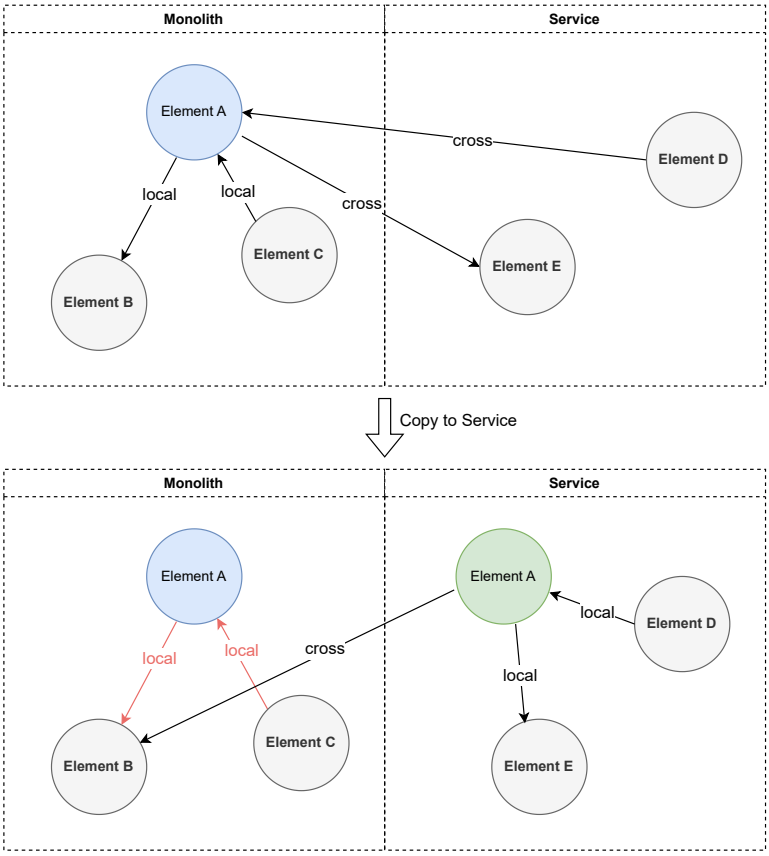

Figure 3.4: Movement to elements and dependencies in a *copy* decision. Cross-boundary dependencies become local to the service. Dependencies from the copy to the monolith become cross-boundary, but dependencies of the original element from/to the monolith remain local.

3.5.2 Library code

When the source code of interest relies on external libraries, the developer must also make decisions regarding the library code. Unlike source code decisions, where flexibility in granularity is essential, extraction decisions for libraries are made as a whole. In other words, the service either depends on the library entirely or does not depend on it at all.

3.5.3 Test code

The system may include test code to validate the feature being extracted. When extracting a feature, the developer aims to extract the relevant tests along with it. Typically, in Java systems, test code is organized to mirror the structure of the main source code, requiring a similar level of flexibility in decision granularity. This includes decisions at test suite-level and test case-level.

In our prototype, our primary focus is on source code and library code, and we differentiate between these two types of code by providing distinct levels of decision granularity. As well, the granularity we choose methods to be the lowest level of source code granularity (thereby excluding fields) to maintain simplicity of dependency analysis. We consider extensions to test code and field-level granularity as potential future work.

Chapter 4

The Extract Prototype

We describe here the components of the prototype and how they contribute to the design goals from Section 2. The prototype is developed as a Java application, featuring a user-facing React frontend. The Java backend takes as input a JAR file of the monolithic application, and generates a dependency graph of the software elements. In this version of the prototype, we rely exclusively on call graph information to generate the dependency graph. However, future iterations should expand this analysis to encompass other types of dependencies including inheritance, implementation, parameter, and return dependencies.

The React frontend interacts with the backend by requesting a partial dependency graph, based on the initial microservice design provided by the developer. We provide an overview of how the frontend visually presents this information and the interactions it supports.

4.1 Dependency Matrix

The user interface presents a matrix-based view of dependencies, where microservice elements are arranged in columns, and monolith elements in rows. order of elements in rows and columns follows a hierarchical structure based on package and class containment (DG6, DG7). Source code elements are presented with method level granularity, permitting for detailed analysis of what is contributing to the dependency. Non-source code elements from external libraries are also shown in the

matrix, but at the granularity of the entire library. This design choice treats libraries as black boxes and helps differentiate source code elements from non-source code elements.

To initiate the extraction task, the developer first selects a set of elements (packages, classes, methods) that constitute the initial microservice seed. These *service methods* are presented in rows, while the columns display all elements that either depend on or are depended upon by the service methods. We refer to the column elements as the *monolith methods*. The cells of the matrix contain dependency information for the intersecting service and monolith methods (Figure 4.1-1). Specifically, they represent *direct outgoing*, *direct incoming*, *transitive outgoing*, and *transitive incoming* dependencies (DG2).

4.2 Making Extraction Decisions

Extract offers decision-making capabilities at different levels of granularity, including method-level decisions (DG1, DG4). This provides the developer flexibility in customizing the scope of decisions, allowing for precise specification of the resulting service. Each column element supports *move to service*, *confirm in monolith*, and *copy to service* decisions, while each row element supports *move to monolith* and *confirm in service* decisions (Figure 4.1-5). These decisions correspond to the actionable extraction decisions introduced in Section 3. Once a decision is made, the element is marked for visual feedback and the dependency matrix updated accordingly. This ensures that the developer can track and visualize the impact of their decisions on the matrix (DG5).

4.3 Summary

For each row and column, we display summarized dependency information: the *column summary* (Figure 4.1-2), and the *row summary* (Figure 4.1-3). A summary cell comprises four values, representing aggregate counts of dependencies in the following categories:

1. Dependencies from other service elements to the element.

2. Dependencies from monolith elements to the element.

3. Dependencies from the element to other service elements.

4. Dependencies from the element to monolith elements.

The summary information provides a quick overview of the degree of coupling an element exhibits between the service and the monolith. Upon hovering any of the summary values, a popup appears, showing the specific elements contributing to the value (Figure 4.1-4).

4.4 Heuristics

The prototype supports five heuristics designed to inform developers in making extraction decisions effectively. The goal of the heuristics is to simplify the identification of elements that the developer should focus on next in the extraction task. They can pinpoint elements that are potentially misplaced in the service design (DG5).

In general, elements that are well-suited for inclusion in the service are highlighted in green, while elements that are more likely to belong to the monolith are marked in red. These visual cues aid developers in discerning the potential placement of each element, facilitating the decision-making process.

The five heuristics supported are:

H1: ***Move to service***:
If a monolith element only depends on service, it likely belongs in the service instead.

H2: ***Move to monolith***:
If a service element only depends on monolith, it likely belongs in the monolith instead.

H3: ***Confirm in service***:
If a service element only depends on the service, it likely does belong in the service.

H4: ***Copy to service***:
If an element is similarly coupled to both service and monolith, then it likely

belongs in both.

$$\frac{FromService + ToService}{Total} \approx_{\alpha} \frac{FromMono + ToMono}{Total} \tag{4.1}$$

where α is the threshold value

H5: ***Confirm in monolith***:

If an element has higher coupling to monolith than service, it likely belongs in the monolith.

$$\frac{FromService + ToService}{Total} <<_{\alpha} \frac{FromMono + ToMono}{Total} \tag{4.2}$$

where α is the threshold value

H1-H3 are all straightforward and definitive heuristics used to determine the inclusion and exclusion of elements. If `methodA` within the monolith relies on elements `dep1`, `dep2`, and `dep3`, all of which are already included in the service, H1 highlights `methodA` green. Likewise, if `methodA` was a service element, H3 would mark it in green to confirm its inclusion within the service. On the other hand, if `methodA` existed in the service, and its dependencies `dep1`, `dep2`, and `dep3` were monolith elements, H2 would highlight `methodA` in red.

H4 and H5 are both heuristics relative to the chosen threshold value. Thus, their accuracy vary depending on the α value and the target monolithic system. To identify copy candidates, H4 evaluates the proportions of service dependencies to the overall number of dependencies, as well as monolith dependencies to the total count. When the difference in ratios falls within the defined threshold α, H4 classifies the element as possessing comparable coupling with both the service and the monolith. A higher value of α would render this heuristic more lenient (identifying more elements for copying), while a lower α would make it stricter. H5 operates in a similar manner by contrasting the identical ratios, but looks for elements that have significantly different values (i.e., more than α) as *confirm in monolith* candidates. Having a higher α value makes the heuristic stricter (permitting only elements with significantly more monolith dependencies than service dependencies), while a lower value makes it more lenient.

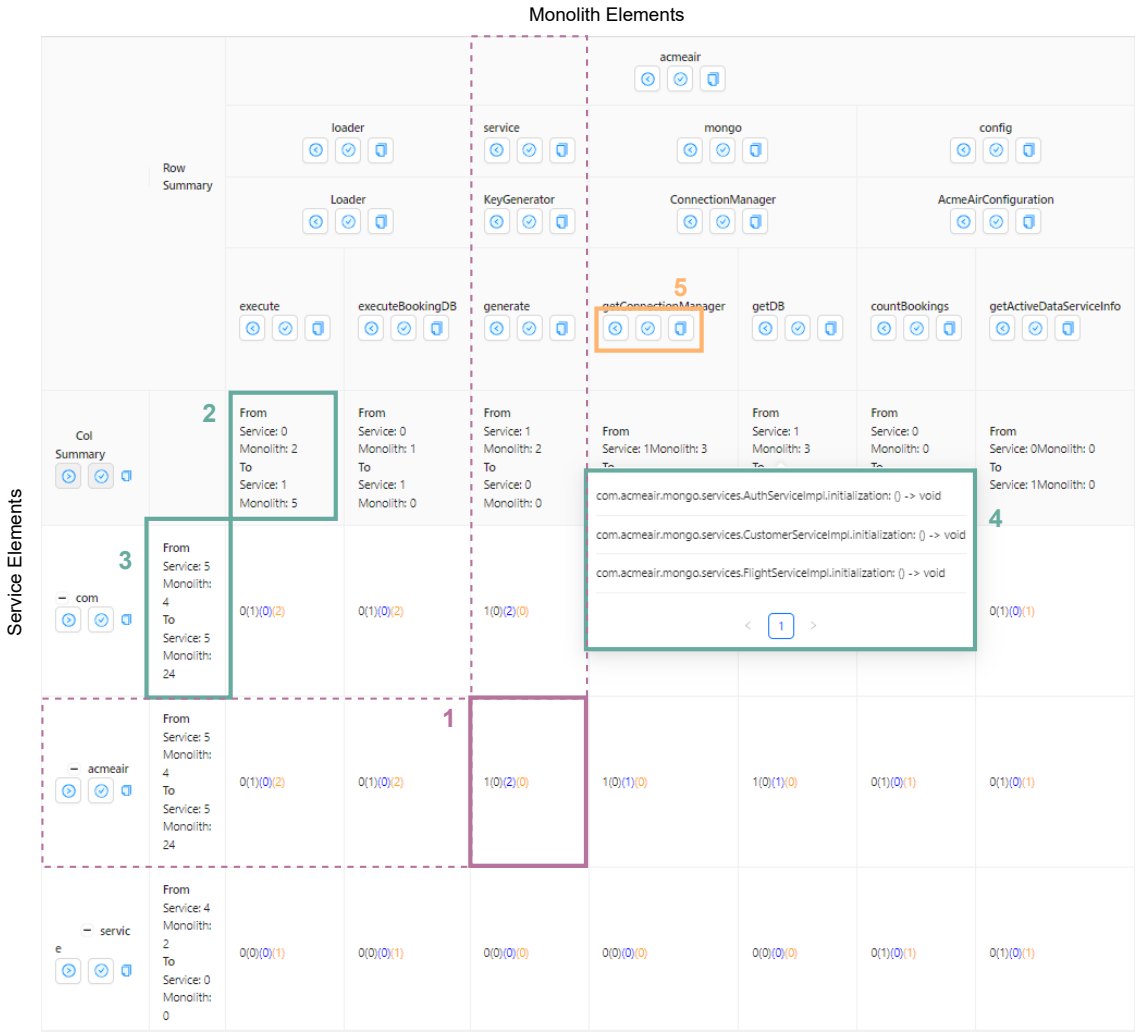

Figure 4.1: A screenshot of the hierarchical dependency as presented to the developer.

Chapter 5

Case Studies

We performed a total of four case studies to illustrate how the prototype is used to decompose Java monolithic systems. Each case study involves a service extraction task where a particular feature, characterized by the developer-input initial service design, is extracted from the monolith. We describe each monolith used and extraction task of the case studies below. For our case studies, we excluded the test code from our task, focusing only on the source code.

The aim of the case studies is two-fold. First, they serve as illustrative scenarios of using the Extract prototype against sample monolithic system. Second, through the case studies we investigate the following research questions regarding the prototype and microservice extraction tasks.

Research Questions

- RQ1 What are the drivers for each extraction decision? Do the heuristics support the decision made?
- RQ2 What is the criteria or indicators that signal the completion of an extraction task?
- RQ3 What factors impact the difficulty in making an extraction decision?
- RQ4 How do the tool-supported microservice designs compare to manually performed designs?

5.1 AcmeAir

AcmeAir[1] is a Java implementation of a fictitious airline management system with the responsibilities to manage web API requests and communicate with the underlying MongoDB database. In the realm of monolith decomposition, AcmeAir stands out not only as a prominent candidate for evaluating decomposition tools but also as a representative instance of a conventional Java application organized by features, making it an ideal choice for our case study. AcmeAir consists of 2,414 lines of source code (LOC), 30 classes, 3 interfaces, and 180 methods. The monolith is organized in a 3-tier architecture with the REST API tier, business logic tier, and database management tier. Each tier is further subdivided for each supported business functionality: `Authentication`, `Booking`, `Customer` management, and `Flight` management. This means locating a feature requires looking across the tiers, but within each tier, the pertinent code is typically colocated. In addition, there are utility code shared by each functionality for loading system configurations, initiating MongoDB connection, and serving the REST APIs. We perform two extraction tasks (ET) involving AcmeAir monolith. We randomly select two features to be extracted in ET1 and ET2.

5.1.1 ET1: Booking Service

The first task aims to extract all code involved in booking management as a single microservice. To begin, the developer makes an initial booking service design by selecting classes and methods they consider to be part of the microservice. Four classes were pre-selected for this purpose.

- `com.acmeair.web.BookingREST`
- `com.acmeair.service.BookingService`
- `com.mongo.services.BookingServiceImpl`
- `com.acmeair.loader.BookingLoader`

[1] https://github.com/blueperf/acmeair-monolithic-java

5.1.2 ET2: Authentication Service

The second task, authentication service involves all code implementing the management of user sessions. As with the `Booking` feature, four classes were preselected as initial service design.

- `com.acmeair.web.LoginREST`
- `com.acmeair.service.AuthService`
- `com.mongo.services.AuthServiceImpl`
- `com.acmeair.loader.SessionLoader`

5.2 DayTrader

DayTrader[2] is an end-to-end benchmark and performance sample application as an online stock trading system, with the capability for users to login, view their portfolio, lookup stock quotes, and buy and sell their shares. It is another extensively utilized Java monolithic application for assessing tools related to monolith decomposition, and unlike AcmeAir which packaged source code elements by features, it demonstrates package-by-layer organization. We select DayTrader for ET3 and 4 in these regards. The monolith is built on Java EE 7, using WebSockets for client-server communication and JDBC for SQL database connection. We focus on the module `daytrader-ee7-ejb` for our case studies, where most application logic exist. The module consists of 6,550 LOC, 27 classes, 4 interfaces, and 493 methods. Compared to AcmeAir, the features of DayTrader are more coupled within the source code. The entry-points to its functionality coexists in `TradeService` interface, implemented by `TradeDirect`, `TradeSLSBRemote` and `TradeSLSBLocal`, which are three different modes supported by DayTrader. The three classes each provide implementations for the supported features including `Quote` retrieval, `Account` management, and `Portfolio` management. We perform additional two extraction tasks involving DayTrader monolith. Following earlier case studies, we randomly select two features from the available choices for ET3 and 4.

[2]https://github.com/WASdev/sample.daytrader7

5.2.1 ET3: Account Service

The `Account` feature contains methods servicing user registration, login and logout, and retrieving and updating account information. The following methods from `TradeService` and its implementers were selected in the initial design.

- `TradeService`, `TradeAction`, `ejb3.TradeSLSBBean`:
  - `login`
  - `logout`
  - `register` (2 overloaded methods for `TradeAction`)
  - `getAccountData`
  - `getAccountProfileData`
  - `updateAccountProfile`
- `direct.TradeDirect`:
  - `login`
  - `logout`
  - `register`
  - `getAccountData` (4 overloaded methods)
  - `getAccountProfileData` (3 overloaded methods)
  - `updateAccountProfile` (2 overloaded methods)
  - `getAccountDataFromResultSet`
  - `getAccountProfileDataFromResultSet`

5.2.2 ET4: Quote Service

The `Quote` feature implements functionalities for creating and retrieving stock quotes. The following methods were preselected in the initial `QuoteService` design.

- `TradeService`, `TradeAction`
  - `getAllQuotes`

  - `getQuote`
  - `createQuote`
  - `updateQuotePriceVolume`
- `direct.TradeDirect`
  - `getAllQuotes`
  - `getQuote` (2 overloaded methods)
  - `createQuote`
  - `updateQuotePriceVolume` (2 overloaded methods)
  - `getQuoteData`
  - `getQuoteForUpdate`
  - `getQuoteDataFromResultSet`
  - `updateQuotePriceVolumeInt`
  - `publishQuotePriceChange`
- `ejb3.TradeSLSBBean`
  - `getAllQuotes`
  - `getQuote`
  - `createQuote`
  - `updateQuotePriceVolume`
  - `publishQuotePriceChange`

5.3 Procedure

In each case study, we documented the extraction decisions, including the chosen software element, the reasoning behind the decision, and the order in which the decisions were made. For each software element involved in the decision, we also recorded the relevant lines of code. These lines of code provided a means to estimate the size of the resulting microservice design. Each task was undertaken by an author as the participant. Furthermore, we performed each extraction task twice, first without the heuristic support and then with heuristics. In the second attempt,

the participant replicated the same extraction while recording any heuristic in support of the decisions. During the case study, a threshold of 0.5 was used for H4 and 0.7 for H5. No time constraints were imposed to allow the participant to complete each task fully. This ensured that we could confirm the end condition of an extraction task (RQ3). Throughout the process, the participant had access to the source code of each monolith.

5.4 Results

Below we describe the results of the case studies. For illustrative purpose, we present a detailed overview of the decisions made and the reasoning behind them for both ET1 and ET3. However, for the second case study involving each monolithic system (ET2, ET4), we present a concise summary of the results.

5.5 ET1: Booking Service

ET1 involved a total of 8 extraction decisions, which comprised 4 *copy*, 3 *move to service*, and 1 *confirm in monolith* decisions. Table 5.1 provides a summary of each decision made, the target of the decision, and lines of code involved in the decision. Figure 5.1 illustrates the movement of elements and dependencies before and after the extraction decisions.

From the initial service seed, the participant noticed a significant imbalance in dependencies, with 24 dependencies from the service to the monolith and only 5 dependencies from the monolith to the service. To minimize the cross-boundary dependencies, the participant studied the row summaries and detail popups to identify the major contributors to the high dependency from the service to the monolith. They found that calls to the external library `org.bson.Document` accounted for 6 out of the 24 dependencies. Based on this information and the column summary of the library, the participant decided to *copy* the `org.bson` library to the service, as two other elements in the monolith also depended on it. A *copy* decision was also made for `com.mongodb` library based on a similar rationale.

These decisions effectively reduced the dependencies from the service to the monolith to three —1/8 of the initial design. The participant continued to reduce the dependencies further, aiming to eliminate the circular dependency between the monolith and the service. One of the remaining dependencies was a call to `services.KeyGenerator.generate` method in the monolith from `mongo.services.BookingServiceImpl.bookFlight`. By looking its column summary, the participant decided to *copy* this method, as it was used by other elements in the monolith but did not have any dependencies back to the monolith. The participant also referred to the source code of the method to ensure that copying it did not transitively require copying all its dependencies. The remaining

two dependencies from the service to the monolith involved calls to methods in the `mongo.ConnectionManager` class. Using the same reasoning, the participant decided *copy* is an appropriate decision.

With the outgoing dependencies to the monolith reduced to zero, the participant then tried to minimize the incoming dependencies. It was found that the service method `loader.BookingLoader.dropBookings` was being called from two elements in the monolith: `loader.Loader.execute` and `loader.Loader.executeBookingDB`. The participant decided to *move* `executeBookingDB` method to the service since its name suggested inclusion to the Booking service, and it only had one dependency from the monolith (ensuring the *move* decision would not increase cross-boundary dependency). However, the participant chose to *confirm* the `execute` method in the monolith, as it had higher coupling with the rest of the monolith (2 incoming, 5 outgoing).

Moving `executeBookingDB` caused a new monolith element, `loader.Loader.clearBookingDB` to depend on the service. After confirming that `clearBookingDB` had zero dependencies to the monolith, the participant made a *move* decision for this method as well. Similarly, the methods `config.AcmeAirConfiguration.countBooking` and `config.AcmeAirConfiguration.getActiveDataServiceInfo` in the monolith called methods in `service.BookingService` of the service, but had no dependency back to the monolith. Consequently, the participant applied the *move to service* decision for these methods as well.

In the resulting design of `Booking` service, the monolith depends on `loader.BookingLoader.dropBookings` of the service, which the participant should surface as service API. By considering the lines of code (LOCs) of all copied and moved elements, we estimate that 61 additional lines of code were introduced, along with two libraries: `org.bson` (238KB) and `com.mongodb` (1.6MB) in JAR sizes, respectively.

5.6 ET2: Authorization Service

During ET2, the participant made a total of 11 extraction decisions, consisting of 5 *copy*, 4 *move to service*, and 2 *confirm in monolith* decisions. Table 5.2 provides

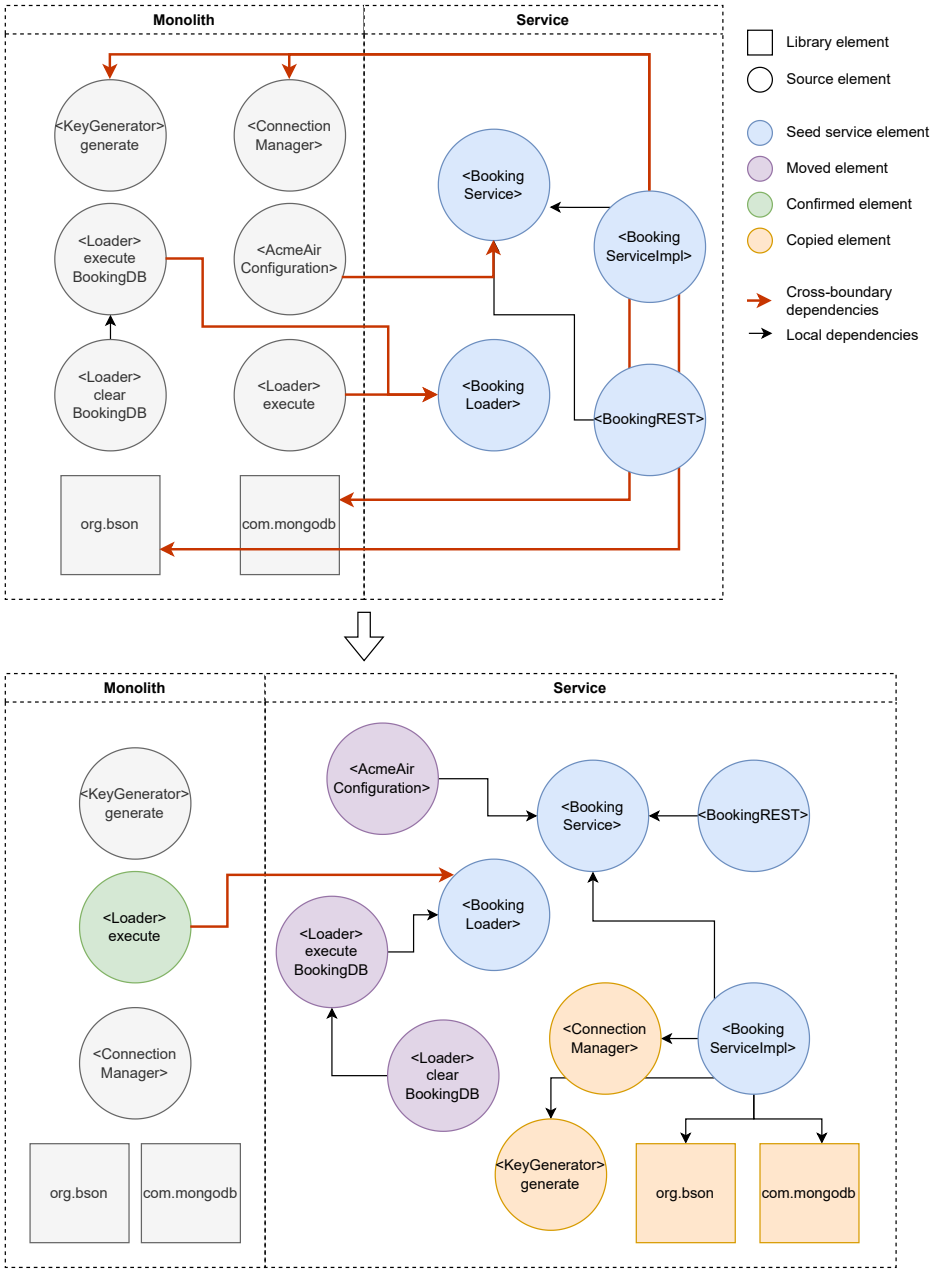

Figure 5.1: Initial `Booking` service seed and the resulting design after extraction decisions

#	Decision	Element Type	Element	LOC
1	C	Library	`org.bson`	—
2	C	Library	`com.mongodb`	—
3	C	Method	`com.acmeair.services.KeyGenerator.generate`	3
4	C	Class	`com.acmeair.mongo.ConnectionManager.*`	13
5	MS	Method	`com.acmeair.loader.Loader.executeBookingDB`	16
6	CM	Method	`com.acmeair.loader.Loader.execute`	9
7	MS	Method	`com.acmeair.loader.Loader.clearBookingDB`	3
8	MS	Class	`com.acmeair.config.AcmeAirConfiguration.*`	26

Table 5.1: The order and type of decisions made in extracting the `Booking` feature as a service.

a summary of the order, target element, and size (LOC) of each decision made. Figure 5.2 illustrates the movement of elements and dependencies before and after the extraction decisions.

As in ET1, the participant completed the task once decisions were made for all monolith (column) elements. In the resulting service design, the participant successfully reduced cross-boundary dependencies to 2, but a circular dependency remained. The monolith called to `loader.SessionLoader.dropSessions` in the service, while the service called to `service.CustomerService.validateCustomer` in the monolith. The participant deemed this circular dependency acceptable since the monolith was in an intermediary stage of full decomposition. The participant expected that the `Customer` feature would be of interest for future extraction, and eventually, the dependency from the `Authorization` service could be redirected to the `Customer` service.

The change in service size from the initial service seed was an additional 95 LOC, 23KB from `org.json` library, `238KB` from `org.bson`, and 1.6MB from `com.mongodb`.

5.7 ET3: Account Service

Compared to ET1 and ET2 where the initial service design comprised coarse-grained class elements, the services extracted from DayTrader in ET3 exhibited method-level granularity. This finer granularity was a consequence of the high interdependence among features in the system, necessitating a greater need for

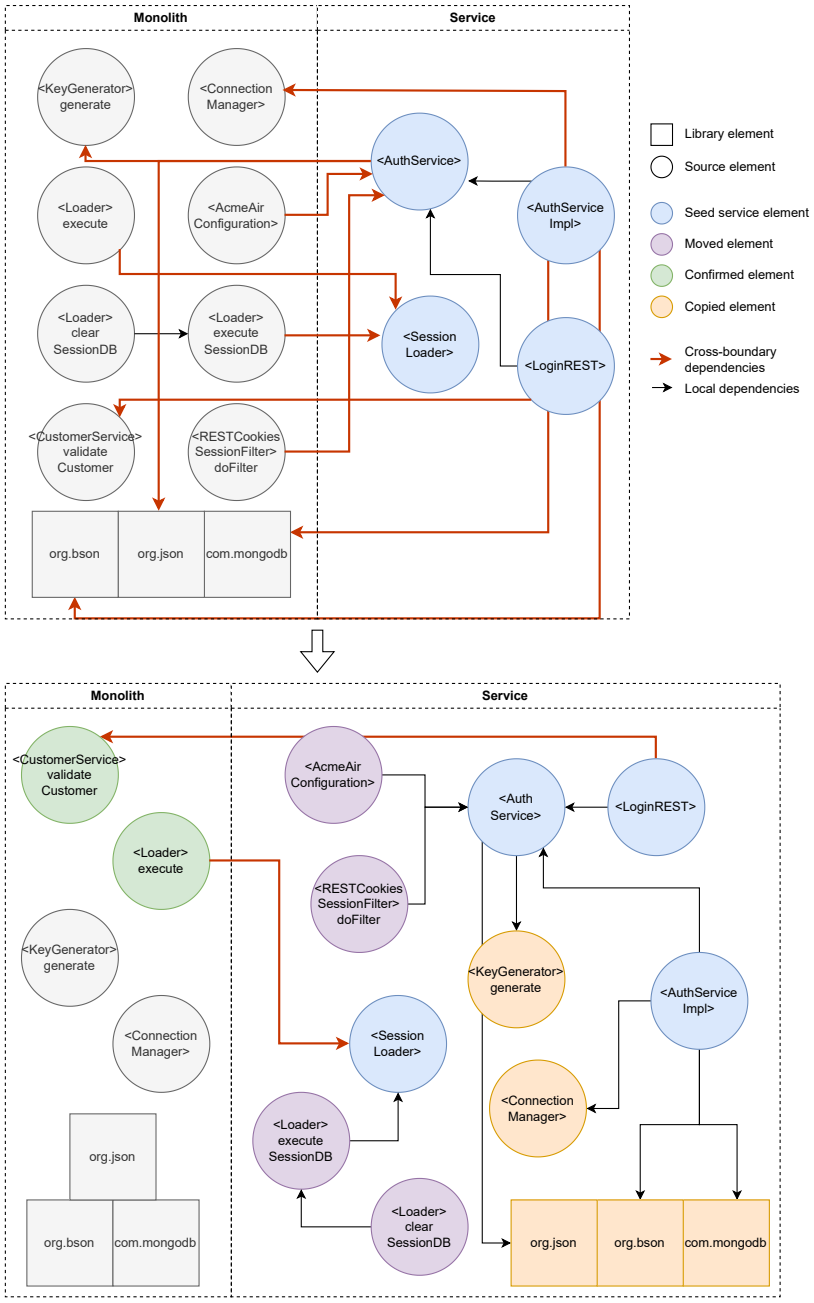

Figure 5.2: Initial `Authorization` service seed and the resulting design after extraction decisions

#	Decision	Element Type	Element	LOC
1	C	Library	`org.bson`	—
2	C	Library	`org.json`	—
3	C	Library	`com.mongodb`	—
4	C	Method	`com.acmeair.services.KeyGenerator.generate`	3
5	C	Class	`com.acmeair.mongo.ConnectionManager.*`	13
6	MS	Method	`com.acmeair.web.RESTCookiesSessionFilter.doFilter`	45
7	MS	Method	`com.acmeair.loader.Loader.executeSessionDB`	16
8	MS	Method	`com.acmeair.loader.Loader.clearSessionDB`	5
9	CM	Method	`com.acmeair.loader.Loader.execute`	9
10	MS	Method	`com.acmeair.config.AcmeAirConfiguration.-countCustomerSession`	13
11	CM	Method	`com.acmeair.services.CustomerService.-validateCustomer`	13

Table 5.2: The order and type of decisions made in extracting the `Authorization` feature as a service.

flexibility when defining its service design.

The participant executed a total of 10 extraction decisions, comprising 2 *confirm in service*, 7 *copy*, and 1 *confirm in monolith* decisions. Table 5.3 summarizes the result of ET3. Figure 5.3 illustrates the movement of elements and dependencies before and after the extraction decisions.

Due to the large number of seed elements in the service, the participant had to expand and delve into the rows to analyze the individual contributions of each package and class. The participant first noted that the class `TradeServices` in the service only had dependencies within the service itself, leading to a *confirm in service* decision.

Next, the participant observed from the row summaries a substantial number of outgoing dependencies from `TradeActions.*` and `direct.TradeDirect.*` methods to `util.Log.*` in the monolith. After confirming that the dependencies of these methods were self-contained within the `Log` class and served purely for event recording functions, the participant decided to *copy* the `Log` class to the service. Row summary and source code of `Log` class were used. Copying the `Log` class also eliminated all the outgoing dependencies from `TradeActions.*` methods to the monolith, leading the participant to make a *confirm in service* decision for `TradeActions.*`.

On the other hand, `direct.TradeDirect.*` still had remaining outgoing dependencies to the monolith. To minimize this dependency, the participant iden-

tified a set of database connection and transaction utility methods as candidates for additional *Copy* decisions. The methods, such as `direct.TradeDirect.commit`, `getConn`, `releaseConn`, `rollBack`, `getInGlobalTxn`, `getStatement`, and `getDataSource`, did not have trailing transitive dependencies, making them suitable for copying. The participant avoided moving these methods as many other elements in the monolith depended upon them, as indicated by the corresponding column summary. Copying the methods of `direct.TradeDirect.*` led to new outgoing dependencies from the service to additional logging methods, which the participant also decided *copy*. This decision reduced the outgoing dependencies from service to the monolith to from 116 to 30.

The participant examined the top-level row summary and found that the majority of remaining dependencies were to `entities.AccountDataProfile` and `entities.AccountData` classes. She concluded that they were data type definitions that probably had practical use in both the service and the monolith. This lead to *copy* decisions being made for both classes.

By applying the same analysis and reasoning, the participant also made a *copy* decision for `KeySequenceDirect.getNextID` method. But the decision introduced extra outgoing dependencies transitively, as the method called `KeyBlock$KeyBlockIterator.hasNext` and `next()` in the monolith. After reviewing the source code and resolving `getNextID`'s transitive dependencies, the participant deemed that copying all the transitive dependencies would be acceptable since it would result in only a slight increase in LOC in the service.

The two remaining outgoing dependencies both involved the `util.TradeConfig` class. The logging methods copied (Decision 2) queried `TradeConfig.getTrace` and `TradeConfig.getActionTrace`, both return boolean values to configure the logs. Even though these two methods were simple to copy (i.e., they were self-contained), the participant chose to *confirm* them in the monolith. Applying the alternative *copy* decisions would have removed the circular dependency between the monolith and the service, but the participant reasoned that `TradeConfig` should eventually be extracted as a service on its own for centralized configuration management service.

As a result, the resulting service design had three incoming dependencies from the monolith to the service and two outgoing dependencies. The service is used by

TradeDirect.buy, sell, and completeOrder, and the two methods they utilize, TradeDirect.getAccountProfileData and getAccountData, need to be surfaced as Account service API. The increase in the size of service from the initial design was 374 LOC.

#	Decision	Element Type	Element	LOC
1	CS	Class	TradeServices.*	6
2	C	Class	util.Log.*	42
3	CS	Class	TradeActions.*	52
4	C	Method	direct.TradeDirect.commit, getConn, releaseConn, rollBack, getInGlobalTxn, getStatement, getDataSource	58
5	C	Method	util.Log.getTrace, getActionTrace, log	9
6	C	Class	entities.AccountProfileDataBean	100
7	C	Class	entities.AccountDataBean	164
8	C	Method	KeySequenceDirect.getNextID	16
9	C	Method	KeyBlock$KeyBlockIterator.hasNext, next	4
10	CM	Method	util.TradeConfig.getTrace, getActionTrace	6

Table 5.3: The order and type of decisions made in extracting the Account feature as a service.

5.8 ET4: Quote Service

ET4 involved a total of 10 extraction decisions, which included 4 *copy*, 3 *move to service*, and 2 *confirm in service* decisions. Table 5.4 summarizes the result of the task. Figure 5.4 illustrates the movement of elements and dependencies before and after the extraction decisions.

In general, the order and choice of decision were similar to those of Account service from ET3. However, unlike the *confirm* decisions made for util.TradeConfig.getTrace and getActionTrace in Decision 5 of ET3, the participant opted to make *move to service* decisions for two other configuration values: getUpdateQuotePrices and getPublishQuotePriceChange. In the former case, the participant noticed from the column summary that the configurations were accessed by other elements of the monolith, while in the latter case, they were exclusively utilized by the service.

The overall increase in the size of the design before and after the task amounted to 230 LOC.

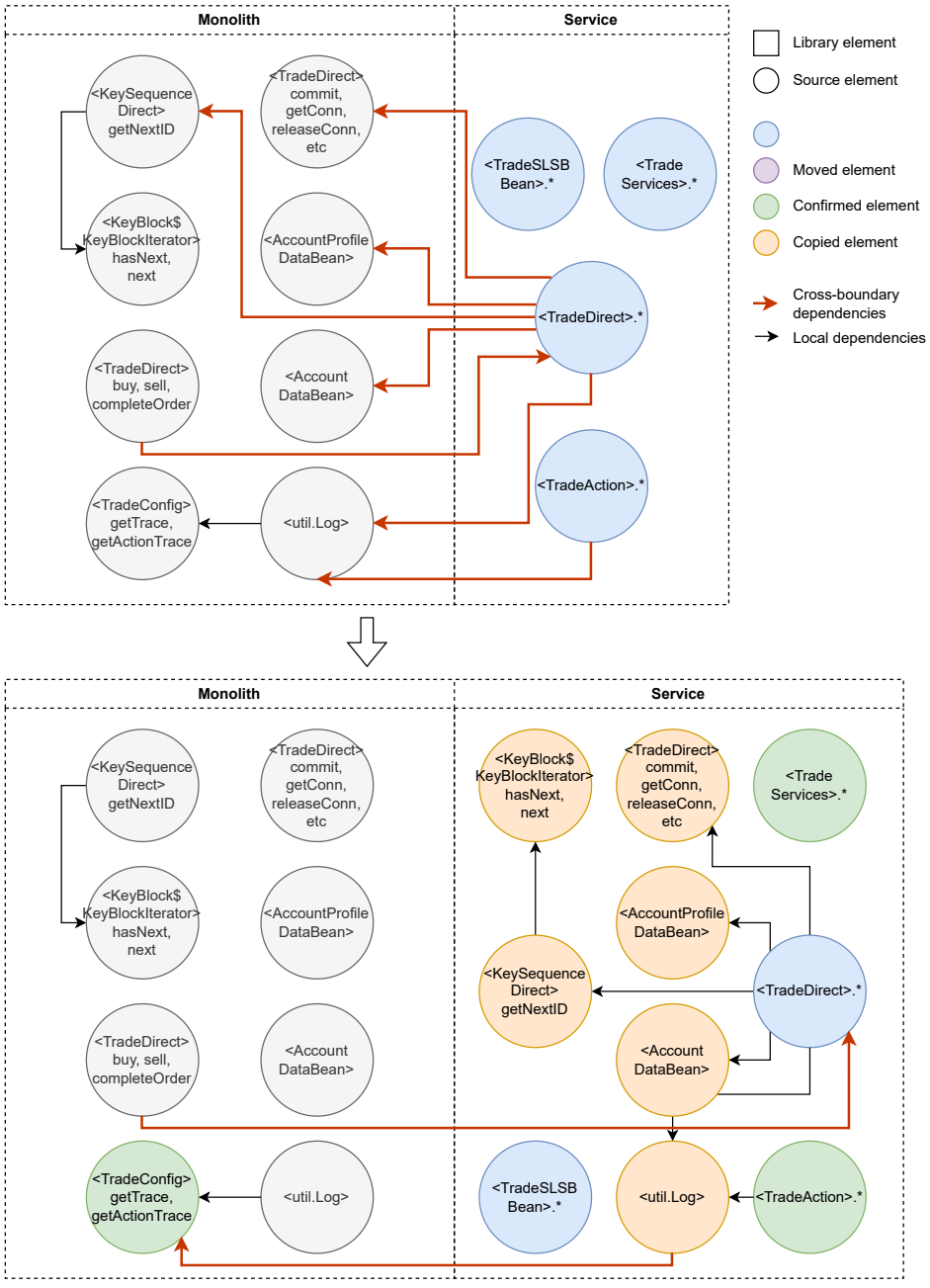

Figure 5.3: Initial `Account` service seed and the resulting design after extraction decisions

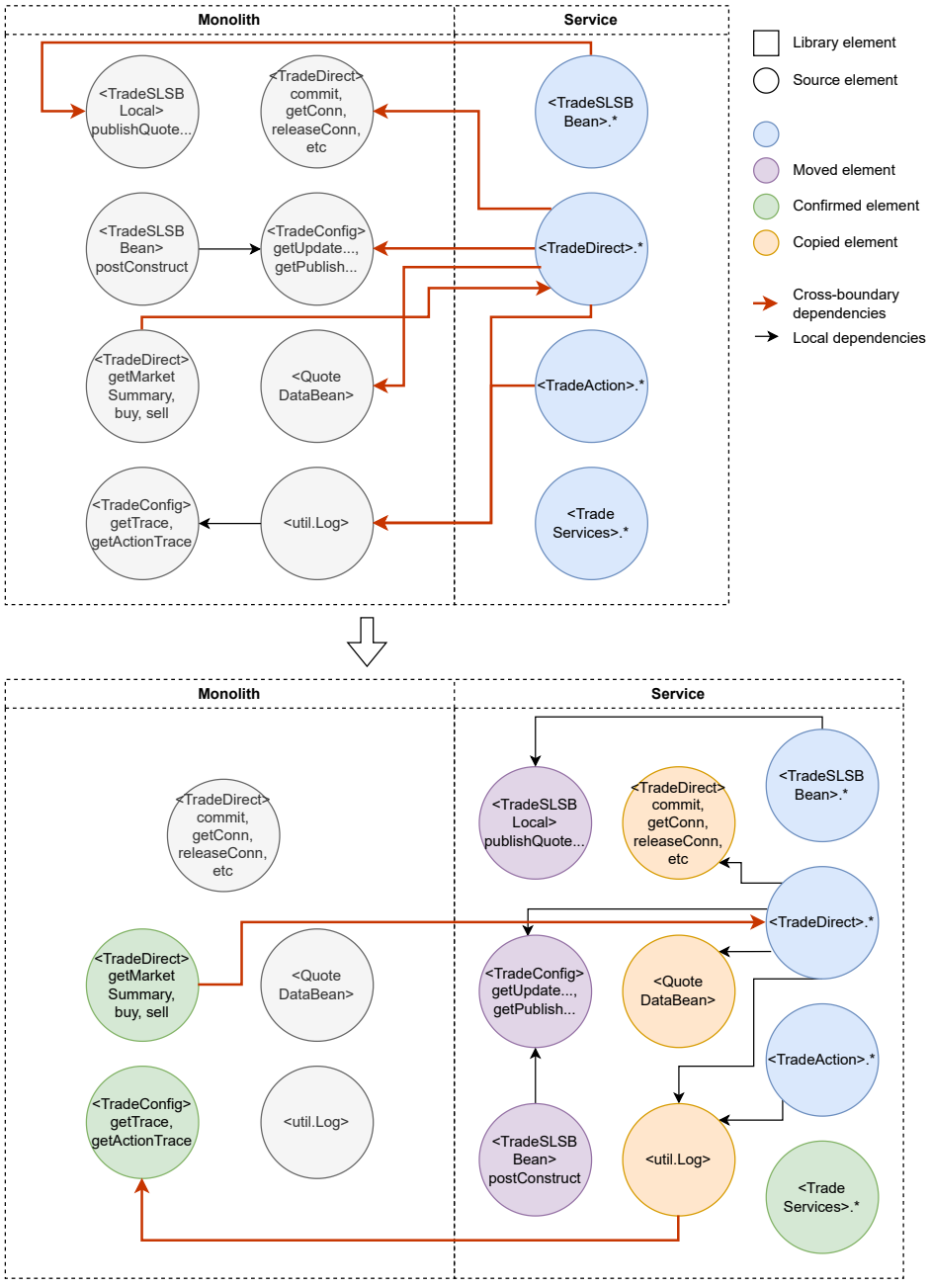

Figure 5.4: Initial `Quote` service seed and the resulting design after extraction decisions

#	Decision	Element Type	Element	LOC
1	CS	Class	`TradeServices.*`	4
2	C	Class	`util.Log.*`	32
3	C	Class	`QuoteDataBean`	117
4	MS	Method	`util.TradeConfig.getUpdateQuotePrices`, `getPublishQuotePriceChange`	6
5	CM	Method	`util.TradeConfig.getTrace`, `getActionTrace`	6
6	MS	Method	`ebj3.TradeSLSBLocal.publishQuotePriceChange`	1
7	MS	Method	`ebj3.TradeSLSBean.postConstruct`	7
8	C	Method	`direct.TradeDirect.commit`, `getConn`, `releaseConn`, `rollBack`, `getInGlobalTxn`, `getStatement`, `getDataSource`	58
9	C	Method	`util.Log.getTrace`, `getActionTrace`, `log`	9
10	CM	Class	`TradeDirect.getMarketSummary`	67

Table 5.4: The order and type of decisions made in extracting the `Quote` feature as a service.

Chapter 6

Discussion

We further analyze the result of case studies and discuss them in terms of our research questions. Our main focus of analysis revolves around identifying factors that either facilitate or hinder decision-making during a microservice extraction task. We focus less on the success or accuracy of the resulting service design from each extraction task, but perform a comparison against manually decomposed microservice designs of the respective monolith.

6.1 RQ1: Extraction Decision Drivers

Research Question I(a)

What are the drivers for each extraction decision?

Table 6.1 presents the participant's rationale behind each extraction decision made while performing the extraction tasks. The decisions are keyed by its task number and order as given by Tables 5.1-5.4. For instance, the eighth decision in extraction task 3 is keyed as ET3-8. Figure 6.1 plots the proportion of rationales behind each extraction decision. The participant did not make a *Move* to service decision in the case studies and so we remove it from this analysis.

As we observe from Figure 6.1, *Copy* decisions are primarily motivated by the target element having a similar degree of coupling to both the monolith and the service. On the other hand, *Move* to service decisions are made when the element in

	Extraction Decision Rationale	Information Used	Decision
R1	A monolith element is depended upon significantly by both the service and the monolith	Column summary	ET1-1, ET1-2, ET1-3, ET1-4, ET2-1, ET2-2, ET2-3, ET2-4, ET2-5, ET3-2, ET3-4, ET3-5, ET3-6, ET3-7, ET3-8, ET3-9, ET4-2, ET4-3, ET4-8, ET4-9
R2	A monolith element depends on few other column elements	Column summary	ET1-3, ET1-4, ET1-5, ET2-4
R3	A monolith element only depends on other service elements	Column summary	ET1-7, ET1-8, ET2-6, ET2-7, ET2-8, ET2-10, ET4-4, ET4-6, ET4-7
R4	A monolith element has more dependency to other monolith elements than to service elements	Column summary, Row summary	ET1-6, ET2-9
R5	A service element only has dependencies within other service elements	Row summary	ET3-1, ET3-3, ET4-1
R6	A monolith element does not belong to the service domain	Column summary, Source code	ET2-11, ET3-10, ET4-5, ET4-10
R7	The element is a data structure	Source code	ET3-6, ET3-7, ET4-3
R8	A monolith element should be extracted out as another service	Column summary, Source code	ET2-11, ET3-10, ET4-5, ET4-10

Table 6.1: Developer's rationale for performing extraction decisions.

monolith exhibits higher coupling to other service elements. Similarly, *Confirm* in service decisions are driven by the element already in service having stronger coupling to other elements within the service. We also observe that *Confirm* can arise from various different rationales, and it is the sole decision requiring the participant to seek extra source code information beyond what Extract prototype provides. The source code information was necessary to recover domain knowledge regarding the target element.

We follow up the question by investigating whether the heuristics support de-

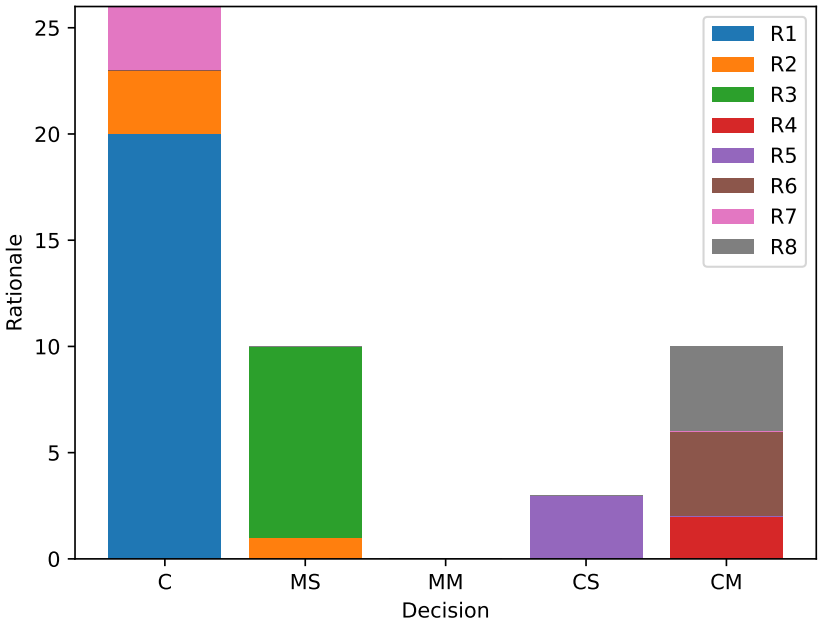

Figure 6.1: Extraction decision versus rationale behind.

velopers in making the intended extraction decisions.

Research Question I(b)

Do the heuristics support the decision?

As described previously in Section 5.3, the participant performed each extraction task twice, once without the heuristic and again with the heuristic. In the second run of the task, the participant replicated the same extraction as in their first attempt, while documenting the heuristic information that backed each of their decisions. Table 6.2 shows the result for each extraction task.

Despite being simple and restrictive, we notice that H1 and 3 identified a significant portion of *move to service*, *confirm in service* decisions. We hypothesize that although the overall extraction task may be complex, when analyzed as individual iterative extraction decisions, each decision often has a straightforward solution.

The threshold-based heuristics, H4, and H5, were also able to identify some of the extraction decisions, but their success varied depending on the task at hand. Increasing H4 threshold qualified more monolith elements for *copy to service* decision, while reducing H5 threshold resulting in more being considered for *confirm in monolith* decision. The developer applied uniform thresholds across all case studies; however, future research should explore how project-specific properties affect the effectiveness of threshold values.

We observed from Figure 6.1 that *confirm in monolith* decisions trace back to a variety of rationales, some of which require further domain and source code knowledge. This need for domain knowledge is also evident in H5's inability to identify certain *confirm in monolith* decisions (e.g. ET2-11, ET3-10, ET4-5, ET4-10).

6.2 RQ2: Task Completion Indicators

Next, we explore the point at which the developer determines that a final service design has been achieved.

Research Question II

What is the criteria or indicators that signal the completion of an extraction task?

The developer was permitted to make the extraction decisions in any order, and determine when the task is complete. Our goal is to identify the conditions indicating an extraction task completion (i.e., the quality of an final service design) by analyzing the reasoning behind the developer's decision to stop making further iterations.

In general, we observe the two goals of microservice extraction, reducing cross-boundary dependencies and eliminating circular dependencies, having a significant influence in the developer's decision to complete a task. For all extraction tasks, the developer proceeded by first reviewing the points where the service relies on the monolith and resolving any unnecessary reliance via the extraction decisions. The review of service's outgoing dependencies was then followed by a review of service's incoming dependencies. The developer completed the task

#	Decision	Heuristic	Correctness
1	C	H4	✓
2	C	H4	✓
3	C	H4	✓
4	C	—	✗
5	MS	H4	✗
6	CM	H5	✓
7	MS	H1	✓
8	MS	H1	✓

(a) ET1: `Booking` service

#	Decision	Heuristic	Correctness
1	C	—	✗
2	C	—	✗
3	C	—	✗
4	C	H4	✓
5	C	—	✗
6	MS	H1	✓
7	MS	H1	✓
8	MS	H1	✓
9	CM	H5	✓
10	MS	H1	✓
11	CM	—	✗

(b) ET2: `Authorization` service

#	Decision	Heuristic	Correctness
1	CS	H3	✓
2	C	H4	✓
3	CS	H3	✓
4	C	H4	✓
5	C	—	✗
6	C	H4	✓
7	C	H4	✓
8	C	H4	✓
9	C	—	✗
10	CM	—	✗

(c) ET3: `Account` service

#	Decision	Heuristic	Correctness
1	CS	H3	✓
2	C	—	✗
3	C	H4	✓
4	MS	H4	✗
5	CM	—	✗
6	MS	H1	✓
7	MS	H1	✓
8	C	H4	✓
9	C	—	✗
10	CM	—	✗

(d) ET4: `Quote` service

Table 6.2: Extraction decisions and correctness of heuristic support.

When she has confirmed all the remaining cross-boundary dependencies, if any, were necessary and intentional. Such judgement often required domain and source code knowledge.

In terms of Extract UI, completion of a task was typically signified by all monolith elements (i.e., columns) being marked as decided. On the contrary, the completion decision was less contingent on whether the row elements were fully decided.

6.3 RQ3: Extraction Decision Challenges

Below we discuss the challenges that developer faced while performing the extraction tasks in the case studies and suggest potential support to can be provided as future work.

Research Question III

What factors impact the difficulty in making an extraction decision?

6.3.1 Limited view of transitive dependencies

The developer focused on immediate cross-boundary dependencies to determine if further iteration is necessary, and if so, which element to target next. However, some extraction decisions required looking beyond the immediate dependencies. For instance in ET1 and 2, the developer decided against the heuristic's suggestion to copy the method `com.loader.Loader.executeSessionDB` but instead moved the element to the service. This decision is justified by the transitive closure of `executeSessionDB`'s dependencies. The dependency chain resolves after making the subsequent decision to move `com.loader.Loader.clearSessionDB`.

The prototype currently does not offer a transparent view of transitive dependencies, specifically, the transitive closure of an element's dependency. To better support making such decisions, the transitive dependencies should be computed and presented for each row and column. This could be accomplished by expanding the summary information to incorporate transitive dependencies and highlighting elements with significant transitive closures.

6.3.2 To copy, to move, or to confirm

When a monolith element was found to be extensively dependent on by service elements, the developer faced the decision of either copying, moving, or confirming the element as is. This decision was usually more straightforward for library elements and more involved for source code elements. For source code elements, the decision to confirm an element required the most analysis beyond the knowledge presented by Extract. *Confirm in monolith* was chosen when the element was

considered unsuitable for the service; i.e. when its semantics belonged outside of the service's domain, or when the developer identified another service it could belong in. Making these observations becomes simpler when the developer have access to element's source code or domain information within the tool UI. Rather than purely focusing on dependency information, future iteration on the prototype should also consider incorporating semantics of the involved elements.

The decision between *copy* and *move* required evaluating the tradeoff between communication frequency and service size. This decision typically involved a monolith element that is wildly utilized by both the service and other monolith elements. When choosing to *copy* an element, the developer improved the separation between the service and the monolith. However, there was a potential downside of increasing the service's size due to including all transitive dependencies of the copied element. The developer would only make the *copy* decision when the increase in service size is justified by the reduction in communication frequency. Making the *copy* versus *move* decision also benefits from further information about the element, including its source code and transitive closure of its dependencies (see 6.3.1).

6.3.3 Large number of interconnected elements

Even though the matrix representation of dependencies mitigated some of the complexity that arise from navigating graph-based information, the visualization still resulted in significant context loss while the developer was making extraction decisions. When presenting a service element with high coupling to the monolith, Extract UI required the developer to scroll across many column elements to review all its dependencies. Likewise, when drilling down on a row element, the developer had to navigate vertically within the table.

To improve the navigability of dependency information within the UI, we can introduce filtering ability to the rows and columns, making the identification of specific software element easier. Moreover, the tool can apply an ordering to the columns based on priority of review or temporarily hide irrelevant columns and rows, focusing only on the software elements in consideration.

6.3.4 Planning the next step in monolith decomposition

Prior to performing any extraction task, an extraction candidate must be identified, along with its starting design. While performing bottom-up, iterative extraction tasks, it is easy to overlook the fact that these tasks collectively contribute to the complete decomposition of the monolith. Strategically selecting the next extraction target is crucial the seamless migration of a monolithic system towards a microservices architecture.

The planning for the next extraction step involves identifying the feature that offers the most benefit while being the easiest to extract. In other word, the developer needs to locate software elements that are highly dependent upon by many other elements and ideally exhibit cohesion. While beyond the scope of a single extraction task, keeping a record of the decisions made during a sequence of extraction tasks could aid in the identification process. For instance, noticing that multiple from ET1 and ET2, we observe that multiple *copy* decisions has been made for `com.acmeair.services.KeyGenerator.generate` element. We could decide to extract this element next in the AcmeAir decomposition process. Retaining decision records from multiple extraction tasks is challenging for a developer, but tooling could make it possible.

6.4 RQ4: Tool-supported vs. Manual Designs

Research Question IV

How do the tool-supported microservice designs compare to manually performed designs?

To answer this question, we compare the designs of each resulting service designs to the corresponding services from manually performed decomposition of AcmeAir[1,2] and DayTrader[3,4].

[1] https://github.com/blueperf/acmeair-bookingservice-java
[2] https://github.com/blueperf/acmeair-authservice-java
[3] https://github.com/sample-daytrader/daytrader-accounts-service
[4] https://github.com/sample-daytrader/daytrader-quotes-service

6.4.1 Elements in tool-supported designs

For each software element in the tool-aided service design, we inspect its presence or absence from the manually extracted service. An element is considered present in the service if a functionally equivalent element exists, irrespective of its code location or exact class/method name. Tables 6.3 and 6.4 provide an overview of this result. Note that the manually decomposed DayTrader currently only support one out of the three modes (`TradeDirect`) supported by the original monolith. We limit our analysis to this particular mode.

For AcmeAir monolith, we observe that `Booking` service has high similarity in terms of the elements being included, with only `clearBookingDB` being excluded from the manual extraction. The implementation of this method solely consisted of a call to `executeBookingDB`, which likely explains its omission during the manual extraction process. On the otherhand, we see a significant difference in the tool-supported and manual designs of `Authorization` service. The manually extracted service contains only the code implementing login and logout functionalities, without any dependency on the MongoDB database. Instead, the service seemed to have underwent further decomposition in which the session management feature was taken out and replaced with JWT[5].

Additionally, we note a notable resemblance in the elements included in the two designs of `Account` and `Quote` services. During manual extraction `TradeServices` interface was replaced with a HTTP controller class and `getInGlobalTxn` was removed as `Account` service had access to its own `Account` database. Similar to the JWT replacement in AcmeAir system, decomposition of DayTrader also modified `util.Log.*` to utilize Log4j[6] instead of `java.util.logging` as in the original monolith. The manual designs also omitted certain logging-related configurations by relying on the configurations provided by Log4j.

6.4.2 Elements in manual designs

Next, we examine the elements present in each manually extracted service and determine if functionally equivalent elements also exist in the Extract service design.

[5]https://jwt.io/

[6]https://logging.apache.org/log4j/2.x/

The manually extracted `Booking` service implemented an additional rewards tracking feature which was originally a shared responsibility of `Flight` and `Customer` components in the monolith. Instead of having direct communication between `Flight` and `Customer` services, the developer opted to position `Booking` service as a intermediary that communicates with both `Flight` and `Customer` to manage rewards gained from a particular booking. This is likely to resolve circular dependency in the extracted services. Aside from reward tracking, the manually extracted `Booking` service also implemented a health check feature. Health check is an API exposed by `Booking` service allowing other services to make calls and ensure that the service is live and accessible. The manual `Authorization` service also included a similar health checking feature, and additional code to make existing login, logout implementation compatible with JWT authentication.

In the manual `Account` service of DayTrader, the only additional functionalities included were the `AccountController` class implementing REST endpoints, and modifications to the `util.Log` class to utilize the Log4j library. However, `Quote` service implemented additional market summary feature which the developer decided against extracting during the case study (ET4-10). In the original DayTrader monolith, the market summary feature is managed by a singleton class `MarketSummarySingleton` that depends on `Quote` methods. The developer conducting the manual extraction chose to integrate the market summary with the `Quote` feature instead of extracting it as a separate service, likely due to the feature's relatively small size.

From these comparisons, we see that while the tool-supported service designs generally contain elements that are consistent with the manually extracted services, the tool-supported designs are often incomplete (i.e., missing elements that exist in the manually extracted services). We hypothesize that the difference is due to Extract prototype lacking support beyond a single service extraction task. The developer performing manual monolith decomposition benefits from a comprehensive view of the entire history of previously extracted services, enabling informed decisions that consider the system as a whole.

Element	Present
service.BookingService	✓
service.KeyGenerator.generate	✓
web.BookingsREST	✓
mongo.services.BookingServiceImpl	✓
mongo.ConnectionManager.*	✓
loader.Loader.executeBookingDB	✓
loader.Loader.clearBookingDB	✗
config.AcmeAirConfiguration.*	✓
org.mongodb	✓
org.bson	✓

(a) ET1: `Booking` service

Element	Present
service.AuthService	✓
service.KeyGenerator.generate	✗
web.LoginREST	✓
web.RESTCookiesSessionFilter.doFilter	✗
mongo.services.AuthServiceImpl	✗
mongo.ConnectionManager.*	✗
loader.Loader.executeSessionDB	✗
loader.Loader.clearSessionDB	✗
config.AcmeAirConfiguration.countCustomerSession	✗
org.mongodb	✗
org.bson	✗
org.json	✗

(b) ET2: `Authorization` service

Table 6.3: The elements present in Extract's designs of AcmeAir services and their presence in the manually extracted counterparts.

Element	Present
TradeServices.*	✗
TradeActions	
login, logout, register, getAccountData, getAccountProfileData, updateAccountProfile	✓
direct.TradeDirect	
login, logout, register, getAccountData, getAccountProfileData, updateAccountProfile	✓
commit, getConn, releaseConn, rollBack getStatement, getDataSource	✓
getInGlobalTxn	✗
entities.AccountProfileDataBean	✓
entities.AccountDataBean	✓
KeySequenceDirect.getNextID	✓
KeyBlock$KeyBlockIterator	
hasNext, next	✓
util.Log.*	✗
TradeConfig	
getTrace, getActionTrace	✗

(a) ET3: `Account` service

Element	Present
TradeServices.*	✗
TradeActions	
getAllQuotes, getQuote, createQuote, updateQuotePriceVolume	✓
direct.TradeDirect	
getAllQuotes, getQuote, createQuote, updateQuotePriceVolume, getQuoteData, getQuoteForUpdate	✓
getQuoteDataFromResultSet, updateQuotePriceVolumeInt	✓
commit, getConn, releaseConn, rollBack getStatement, getDataSource	✓
getInGlobalTxn, publishQuotePriceChange	✗
entities.QuoteDataBean	✓
util.Log.*	✓
TradeConfig	
getUpdateQuotePrices, getPublishQuotePriceChange	✓
getTrace, getActionTrace	✗

(b) ET4: `Quote` service

Table 6.4: The elements present in Extract's designs of DayTrader services and their presence in the manually extracted counterparts.

Chapter 7

Threats to Validity

Construct validity The use of lines of code as a measure to estimate microservice size may not fully capture the complexity and functionality of the resulting service designs. Instead of relying solely on LOCs, combining other metrics such as service interactions, coupling and cohesion, and modularity, could better capture the complexity and functionality of the designs.

Internal validity. The author conducted the case studies as the developer, and although the result is based on objective observation, this introduces experimenter bias. In the second extraction attempt for each case study, heuristics were employed, but the developer replicated their initial attempt and merely assessed the heuristics for their indicative ability. This approach minimizes heuristic bias, but the result may still be influenced by the threshold value used. To enhance internal validity, case studies should be conducted by external participants and consider multiple threshold values to identify any consistent patterns or biases.

External validity. The case studies were performed by a single developer, and were limited to evaluating two Java monolithic systems. While the selected monoliths were chosen to showcase different degree of feature collocation and are commonly used in related research, additional studies involving more participants and a broader range of Java monoliths are necessary to enhance the generalizability of the findings.

Chapter 8

Limitations and Future Work

Our approach promotes developers to perform iterative monolith decomposition by focusing on a single microservice extraction task. However we have identified four main limitations to this approach:

1. *Dynamic and data dependencies are not considered.* The approach on static analysis only to retrieve coupling information of the elements relevant in the extraction task. However, alternative approaches commonly integrate runtime and data dependency information into their analysis, allowing developers to obtain a more thorough view of the program. However incorporating these information would also complicate the extraction task as developers would need to analyze and evaluate various types of dependencies.

2. *Non-code elements are not analyzed and can not be extracted.* The approach only considers code as software elements, and non-code artifacts such as documentation, configuration files, and test or release-related files cannot be extracted as part of the extraction task. To overcome this limitation, additional research is needed to define the implicit dependencies between non-code artifacts and code elements.

3. *Relies on developer-input of an initial service design.* The starting point of the approach is an initial microservice design crafted by the developer. This is an manual overhead specific to our approach that is not existent in alternative, more autonomous approaches. Nevertheless, considering another

perspective, this overhead is somewhat deliberate and compels the developer to conduct a pre-analysis for the extraction task. Our approach presupposes that the developer possesses a certain degree of familiarity with the system they are decomposing.

4. *Approach may become an overhead with simpler microservice extraction.* Manual extraction may be more appropriate if the original monolith is highly modular and the feature being extracted is self-contained. During the pre-analysis (for devising an initial plan), The developer may come to realization that the task is straightforward, and proceed to manually extract the service.

Given the limitations of the approach and challenges observed during the case studies, we propose a number of improvements and extensions to our approach and the tool prototype that we see beneficial to research in iterative monolith decomposition.

1. *Integration with IDE.* The case studies uncovered a need for source code reference while performing the extraction tasks, regardless of the dependency information presented by the visualization. Developers can extract essential domain knowledge from the source code. Integrating with an IDE would fulfill this requirement without necessitating additional context switches for the developer.

2. *Enhanced heuristics information.* For the prototype, we introduced five heuristics, which, despite their simplicity, aligned with a substantial portion of the decisions made. New heuristics, such as those that take into account multiple extraction tasks, could be developed to assist developers in making ”good” decisions and avoiding ”bad” ones.

3. *Incorporate test code into extraction task.* We described the different types of software code and their impact on the granularity of decisions made for a specific element (Section 3.5). Of the three types, the prototype accounts for source and library code, but an extension should be made to test code. In general, text code relevant to a feature being extracted also belongs in the resulting service. The extracted test code could also serve the purpose of

asserting that the extraction preserves the behaviors from the original monolith.

4. *Support analysis of wider variety of dependencies.* Dependencies in Java systems are not limited to method calls. Other types of dependencies such as field dependencies, class hierarchy dependencies (e.g., implements, inherits), and input and return type dependencies also contribute to couplings between software elements. For a more thorough analysis, extension to these dependencies would be necessary.

5. *Persist decisions across multiple extraction tasks.* The comparisons between Extract-assisted service design and manually extracted services demonstrated how a decision in one extraction task can influence subsequent decisions in another task. Iterative decomposition benefits the developer by breaking down monolith decomposition into concrete extraction tasks. Nevertheless, a holistic view is crucial to make decisions that align with the overall decomposition.

6. *Provide more space-efficient visualizations.* From the case studies, we observed a higher usage of row and column summaries than the cell values. This is because the current matrix visualization in the prototype replicates dependency information both in the summary popups and the cell values. The cells do provide additional transitive dependency counts but this information was underutilized by the developer. In order to provide a more space-efficient visualization, this duplication should be reconsidered.

7. *Provide automated support for performing code changes.* The current prototype produces a service design for actual extraction to be performed manually by the developer. The tool could be extended to assist the resulting code changes as well, once the service design is approved by the developer. For example, the tool could generate an API layer based on the remaining cross-boundary dependencies between the service and the monolith. Further research into automated source code modification could enable such extension.

Chapter 9

Conclusion

Monolith decomposition is a complex task that brings system-wide changes. We identified a disparity between the decomposition process supported by current state-of-the-art tools and how a developer would usually approach this kind of decomposition. The current automated decomposition tools introduce *big-bang rewrites* to the system, which contradicts with the developer's iterative decomposition workflow. We have presented the Extract prototype for supporting developer better in such iterative decomposition process. Drawing inspiration from research in feature extraction and dependency analysis tools, we specialized Extract for reasoning about a single microservice extraction task. Extract supports the developer in constructing a microservoice design by presenting a focused view of software element dependencies, providing an interface for making *extraction decisions* on each relevant element, and suggesting plausible decisions by means of heuristics.

We performed four case studies which illustrate the usage of the Extract prototype against two open-source monolithic Java systems. The case studies show that using Extract, the participant can produce microservice designs that are consistent with manually performed extractions. By analyzing the type and sequence of extraction decisions in each case study, we provide insights into the factors influencing each decision, the criteria for task completion, and the challenges faced during the extraction process.

Our bottom-up, iterative microservice extraction approach divides monolith decomposition into smaller, tractable tasks that are less intrusive to a live monolithic

application. This approach is applicable to various forms of system decomposition, extending beyond the domain of monolith decomposition and microservices. Extension to the approach should incorporate knowledge from multiple extraction tasks, in order to better inform decisions that span across individual extraction tasks.

Bibliography

[1] C. Brandt and A. Zaidman. Developer-centric test amplification. *Empirical Software Engineering*, 27(96), 2022. doi:10.1007/s10664-021-10094-2. → page 7

[2] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. → page 7

[3] P. Calçado. Building products at soundcloud, part 2: Breaking the monolith, 6 2014. URL https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith. → pages 1, 2

[4] J. M. Carroll. Human-computer interaction: Psychology as a science of design. *Annual Review of Psychology*, 48(1):61–83, 1997. doi:10.1146/annurev.psych.48.1.61. URL https://doi.org/10.1146/annurev.psych.48.1.61. PMID: 15012476. → page 7

[5] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *International Workshop on Program Comprehension*, pages 241–247. IEEE, 2000. → page 6

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003. doi:10.1109/TSE.2003.1183929. → page 6

[7] S. Eski and F. Buzluca. An automatic extraction approach: Transition to microservices architecture from monolithic application. In *Proceedings of the International Conference on Agile Software Development: Companion*, New York, NY, USA, 2018. ISBN 9781450364225. doi:10.1145/3234152.3234195. URL https://doi.org/10.1145/3234152.3234195. → pages 2, 4

[8] R. Falke, R. Klein, R. Koschke, and J. Quante. The dominance tree in visualizing software dependencies. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, 2005. doi:10.1109/VISSOF.2005.1684311. → page 9

[9] M. Fowler. Strangler fig application, 2004. URL https://martinfowler.com/bliki/StranglerFigApplication.html. → page 2

[10] M. Fowler and J. Lewis. Microservices: A definition of this new architectural term. 2014. URL https://martinfowler.com/articles/microservices.html. → page 1

[11] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *International Conference on Software Testing, Verification and Validation*, pages 362–369, 2013. doi:10.1109/ICST.2013.51. → page 7

[12] M. Ghoniem, J.-D. Fekete, and P. Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Symposium on Information Visualization*, pages 17–24, 2004. doi:10.1109/INFVIS.2004.1. → page 9

[13] A. Gluck. Microservice architecture at uber, 7 2020. URL https://www.uber.com/en-CA/blog/microservice-architecture/. → page 1

[14] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302, 2017. doi:10.1109/ICSAW.2017.9. → page 11

[15] I. Herman, G. Melancon, and M. Marshall. Graph visualization and navigation in information visualization: A survey. *Transactions on Visualization and Computer Graphics (TOG)*, 6(1):24–43, 2000. doi:10.1109/2945.841119. → page 8

[16] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4), 2 2013. ISSN 1049-331X. doi:10.1145/2377656.2377657. URL https://doi.org/10.1145/2377656.2377657. → pages 6, 8, 9

[17] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee. Mono2micro: A practical and effective tool for decomposing monolithic

java applications to microservices. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1214–1224, New York, NY, USA, 2021. doi:10.1145/3468264.3473915. URL https://doi.org/10.1145/3468264.3473915. → pages 2, 4, 11

[18] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, and A. Matsuo. Sarf map: Visualizing software architecture from feature and layer viewpoints. In *International Conference on Program Comprehension (ICPC)*, pages 43–52, 2013. doi:10.1109/ICPC.2013.6613832. → page 9

[19] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16, 2020. doi:10.1109/ICSA-C50368.2020.00011. → pages 2, 4, 9

[20] J. Laval and S. Ducasse. Resolving cyclic dependencies between packages with enriched dependency structural matrix. *Software: Practice and Experience*, 44(2):235–257, 2014. doi:https://doi.org/10.1002/spe.2164. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2164. → page 9

[21] T. Mauro. Microservices at netflix: Architectural best practices, 2 2016. URL https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. → page 1

[22] G. Mazlami, J. Cito, and P. Leitner. Extraction of microservices from monolithic software architectures. In *International Conference on Web Services (ICWS)*, pages 524–531, 2017. doi:10.1109/ICWS.2017.61. → pages 2, 4, 5

[23] T. Mens and T. Tourwe. A survey of software refactoring. *Transactions on Software Engineering (TSE)*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817. → page 1

[24] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii. Visualization tool for designing microservices with the monolith-first approach. In *Working Conference on Software Visualization (VISSOFT)*, pages 32–42, 2018. doi:10.1109/VISSOFT.2018.00012. → pages 2, 4, 5

[25] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019. → pages 1, 2

[26] J. O'Hanley. Package by feature, not layer, Jan 2023. URL http://www.javapractices.com/topic/TopicAction.do?Id=205#:~:text=Package%2Dby%2Dfeature%20uses%20packages,with%20minimal%20coupling%20between%20packages. → page 18

[27] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi. Towards microservice smells detection. In *Proceedings of the International Conference on Technical Debt*, TechDebt, page 92–97, New York, NY, USA, 2020. doi:10.1145/3387906.3388625. URL https://doi.org/10.1145/3387906.3388625. → page 15

[28] M. Pinzger, K. Grafenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *International Conference on Program Comprehension (ICPC)*, pages 254–259, 2008. doi:10.1109/ICPC.2008.23. → page 8

[29] F. Ponce, G. Márquez, and H. Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019. doi:10.1109/SCCC49216.2019.8966423. → page 4

[30] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):1–29, 2012. → page 7

[31] C. Richardson. Strangler fig application pattern: Incremental modernization to services, June 2023. URL https://microservices.io/post/refactoring/2023/06/21/strangler-fig-application-pattern-incremental-modernization-to-services.html. → page 2

[32] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3–es, 2 2007. ISSN 1049-331X. doi:10.1145/1189748.1189751. URL https://doi.org/10.1145/1189748.1189751. → page 6

[33] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the the joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 15–24, 2007. → page 6

[34] R. Schaut. Mac word 6.0, 2 2004. URL https://web.archive.org/web/20160412213959/https://blogs.msdn.microsoft.com/rick_schaut/2004/02/26/mac-word-6-0/. → page 1

[35] C. Schröer, F. Kruse, and J. Marx Gómez. A qualitative literature review on microservices identification approaches. In S. Dustdar, editor, *Service-Oriented Computing*, pages 151–168, Cham, 2020. Springer International Publishing. → page 2

[36] K. Sellami, M. A. Saied, A. Ouni, and R. Abdalkareem. Combining static and dynamic analysis to decompose monolithic application into microservices. In J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, and A. Ruiz-Cortés, editors, *Service-Oriented Computing*, pages 203–218, 2022. → pages 2, 4

[37] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools: (tool paper). In *Computer Aided Verification: International Conference, (CAV), August 17-20, 2006. Proceedings 18*, pages 419–423, 2006. → page 7

[38] M. Shahin, P. Liang, and M. A. Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software (JSS)*, 94:161–185, 2014. ISSN 0164-1212. doi:https://doi.org/10.1016/j.jss.2014.03.071. URL https://www.sciencedirect.com/science/article/pii/S0164121214000831. → pages 8, 9

[39] H. P. Singh, V. Kumar, and A. Agrawal. Demystifying information technology projects success and failure factors: The cases of california DMV and bancoitamarati bank. *Mumukshu Journal of Humanities*, 7(1): 493–499, 2015. → page 1

[40] J. Spolsky. Things you should never do, part i, 4 2000. URL https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/. → page 1

[41] S. Staffa, G. Quattrocchi, A. Margara, and G. Cugola. Pangaea: Semi-automated monolith decomposition into microservices. In H. Hacid, O. Kao, M. Mecella, N. Moha, and H.-y. Paik, editors, *Service-Oriented Computing*, pages 830–838. Springer International Publishing, 2021. → pages 2, 4

[42] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. Microservice architecture in reality: An industrial inquiry. In *International Conference on Software Architecture (ICSA)*, pages 51–60, 2019. doi:10.1109/ICSA.2019.00014. → page 2