國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

從需求到微服務：以領域驅動設計與機器學習爲方法
From Requirements to Microservice: A Domain Driven Approach with Machine Learning

林怡伶
Yi-Ling Lin

指導教授：李允中 博士
Advisor: Dr. Jonathan Lee, Ph.D.

中華民國 112 年 7 月
July, 2023

doi:10.6342/NTU202303162

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

從需求到微服務：以領域驅動設計與機器學習為方法

From Requirements to Microservice: A Domain Driven Approach with Machine Learning

本論文係林怡伶君（學號 R10922165）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 27 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 27 July 2023 have examined a Master's thesis entitled above presented by LIN, YI-LING (student ID: R10922165) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

李允中	[illegible]	鄭有進
（指導教授 Advisor）		
徐國勛	陳錫民	

系主任/所長 Director: 洪士灝

誌謝

首先，我要衷心感謝我的指導教授李允中博士過去兩年間的提攜與指導，使我學會系統化的切入問題並分析解決方法，且在過程中給予我寶貴的建議，讓我在學術研究上有所成長。同時，我要感謝實驗室的夥伴：陳力聖、林辰臻、劉仁軒、許恆、張馨尹、梁峻瑞，在修課與研究的過程中相互交流與幫助。再者，我要感謝學弟妹與助理錢怡君，還有家人的幫助，讓我能夠順利完成此篇論文。

doi:10.6342/NTU202303162

摘要

微服務架構已成爲開發應用程式的熱門選擇，因其具有簡潔、可獨立部署和可溝通的服務特性。然而，儘管有指導設計過程的實際範例與原則，但並沒有明確的規則來説明如何根據需求來設計微服務。針對此議題，領域驅動設計（DDD）因提供將領域分解的方法而變得相關，但在實際規劃微服務架構時仍會面臨許多挑戰。這些困難包括直接從領域模型生成微服務或決定微服務的適當粒度。爲了應對這些挑戰，在此篇研究中，我們提出了一個新穎的兩階段流程。該流程不考慮領域模型（domain model），而是利用 DDD 和 event storming 的原則來定義子領域（subdomain）邊界。在子領域的邊界決定之後，我們利用配對機制將這些子領域與潛在的微服務候選者進行映射，以此確保配對是多元並具有彈性的。最後我們使用機器學習來自動化此流程，進而提升系統的效率。

關鍵詞 — 微服務、領域驅動設計、機器學習、需求工程、自然語言處理

doi:10.6342/NTU202303162

Abstracts

The microservices architecture has emerged as a popular choice for developing applications as compact, independently deployable, and conversational services. While there exist proven practices and principles that can guide the design process, there are no definitive rules that dictate how to design microservices based on requirements. Domain-driven design (DDD) has gained relevance as it provides a means for the decomposition of domains into contexts, but challenges arise when applying DDD to plan microservices architecture. Difficulties include deriving microservices directly from domain models or determining the appropriate granularity of microservices. To cope with these challenges, in this work, we propose a novel two-stage process that disregards domain models in the process. It leverages DDD and event-storming principles to define subdomain boundaries. Subsequently, a sophisticated matchmaking mechanism connects these subdomains with potential microservice candidates, offering versatile and flexible mapping. The process is automated using machine learning, enhancing its effectiveness.

Index terms — Microservices, Domain-Driven Design, Machine Learning, Requirements Engineering, Natural Language Processing

doi:10.6342/NTU202303162

Contents

doi:10.6342/NTU202303162

doi:10.6342/NTU202303162

doi:10.6342/NTU202303162

List of Figures

doi:10.6342/NTU202303162

doi:10.6342/NTU202303162

List of Tables

doi:10.6342/NTU202303162

Chapter 1

Introduction

With the advent of cloud computing, traditional system architectures fall short of meeting the demands and requirements of cloud-based environments. Cloud computing allows for dynamic scaling of resources to manage fluctuating workloads, whereas traditional monolithic architectures are typically designed for fixed hardware. Furthermore, cloud computing's infrastructure promotes modularity, which contradicts monolithic applications, in which all components are tightly coupled, presenting challenges when updating or modifying specific functionalities. The cloud also provides better fault tolerance and higher availability through its built-in fault tolerance mechanism, which is often absent in traditional architectures.

On the other hand, microservices, due to their loosely coupled and independently deployable nature, align well with cloud computing requirements and help overcome the limitations of traditional architectures. As a result, microservices have become the new standard for developing highly modularized software systems. While mi-

doi:10.6342/NTU202303162

croservices have gained popularity for their numerous benefits, their adoption can indeed be challenging, whether one chooses to migrate a legacy monolithic application to microservices or design them from scratch. One of the difficulties lies in how to define appropriate boundaries for microservices.

Fortunately, domain-driven design (DDD) [6] helps in planning microservice architecture. By defining subdomain boundaries and designing corresponding domain models that specifically tackle problems with each domain, DDD helps streamline the design process and facilitates the adoption of microservices. While there exist proven practices utilizing DDD principles for successful microservices design, the process of deriving microservices from the designed domain model and determining the appropriate number of microservices remains unclear. To address this issue, we propose a matchmaking mechanism to derive microservices from bounded contexts without relying on domain models.

In addition to the above issue, the DDD design process demands continuous communication and collaboration between the developers and domain experts, which can be time-consuming. We have observed that there is limited research on planning and generating microservices based on system requirements. Requirements, presented in various forms such as plain text, use case diagrams, use case specifications, etc., offer comprehensive information about a planned software system. In this work, we aim to leverage requirements to automate the design of microservices.

More specifically, a two-stage process is proposed in this work to offer assistance in planning microservices based on requirements, reducing the need for human effort. The concepts of DDD are employed to define subdomain boundaries, and a match-

doi:10.6342/NTU202303162

making mechanism is introduced to bridge the gap between these subdomains and the underlying microservice(s). By leveraging machine learning, certain processes can be automated, enhancing efficiency.

The rest of this paper is organized as follows: Chapter 2 introduces the related works; Chapter 3 provides an overview of the proposed two-stage process; Chapter 4 describes the first stage, where bounded contexts are derived from requirements; Chapter 5 details the second stage, where bounded contexts are mapped to their corresponding microservices; Chapter 6 demonstrates the user interface and explains how users can utilize the proposed method; Finally, Chapter 7 summarizes this research's contributions and Chapter 8 outlines the future work.

doi:10.6342/NTU202303162

Chapter 2

Related Work

2.1 Background Work

2.1.1 Microservices

In a separate study [3], Chen decomposes open-source monolithic applications into reusable microservices, leading to the construction of a pool of microservice candidates. The pool of microservice candidates resulting from this decomposition process serves as a valuable resource for our research. Moreover, according to Chen's definition, microservices encompass four key elements: controller, service, repository, and entity.

In our approach, we leverage the descriptions corresponding to these four elements to match each of the bounded contexts obtained from the requirements to the most suitable microservices from the candidate pool. This process allows us to

doi:10.6342/NTU202303162

successfully attain our objective of deriving microservices from the requirements.

2.1.2 Domain-Driven Design (DDD)

Domain-driven design (DDD) is a concept introduced by Eric Evans [6]. It is an approach to software development that focuses on building a rich and meaningful representation of the domain within the software. It encompasses two essential phases: strategic design and tactical design. The strategic design phase involves understanding the business domain and defining bounded contexts for the domain models. On the other hand, during the tactical design phase, the tactical patterns are applied to create coherent domain models that represent the business domain. In this work, we adhere to the concepts and definitions defined in DDD to design the overall process, ensuring that the derived microservices align with the core aspects of the business domain. Table 2.1 includes some significant DDD definitions that are leveraged in this work.

2.1.3 Event Storming

Event Storming is an interactive and collaborative meeting technique specifically tailored for DDD [6]. Through its visual and engaging process, it encourages team members to collaborate and share their knowledge, thereby uncovering domain events and identifying business rules. During an Event Storming session, team members use sticky notes as a visual representation of various concepts. These sticky notes are placed on a large whiteboard or wall, illustrating the flow of the business

doi:10.6342/NTU202303162

DDD Concept	Definition
Domain	A sphere of knowledge, influence, or activity.
Bounded context	The delimited applicability of a particular model. Bounded contexts give team members a clear and shared understanding of what has to be consistent and what can develop independently.
Model	A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
Entity	A representation of an object in the domain. It is fundamentally defined not by its attributes, but by a thread of continuity and identity.

Table 2.1: DDD definitions relevant to this work.

process. The concepts gathered during an event storming session fall into several categories, each designated with a unique color of sticky note, as shown in Table 2.2.

Event storming involves four main steps:

1. Identify domain events and place all the event sticky notes in sequence on a timeline.
2. Add the commands and actors that caused the domain events.
3. Identify aggregates by grouping command and event pairs that are related.
4. Define bounded contexts by drawing boundaries based on business logic and the relationships between events and aggregates.

In our research, we utilize the fundamental principles of these four steps to derive bounded contexts. However, we introduce modifications to adapt the process

doi:10.6342/NTU202303162

to our specific input requirements. To avoid introducing further ambiguity, we refrain from distinguishing aggregates and entities. Instead, our approach focuses on discerning the differences between entities and attributes, which simplifies the derivation process.

Event Storming Concept	Definition
Domain event	An event that occurs in the business process. Written in the past tense.
Actor	A person who executes a command through a view.
Command	A command executed by a user through a view on an aggregate that results in the creation of a domain event.
Aggregate	Cluster of domain objects that can be treated as a single unit.

Table 2.2: Event storming concepts with their corresponding color and definitions.

2.1.4 EARS Requirements Template

The Easy Approach to Requirements Syntax (EARS) [9] is an effective method of expressing requirements. The types of EARS requirements and their corresponding format are shown in Table 2.3. To structure our input requirements, we adopt the EARS template, which provides a clear and organized framework for capturing and presenting the necessary information.

2.1.5 ConceptNet

ConceptNet [11] is a freely accessible large-scale commonsense knowledge base with an integrated NLP toolkit that supports many practical textual-reasoning tasks

doi:10.6342/NTU202303162

Type of EARS requirements	Pattern
Ubiquitous requirements	The <System name> SHALL <System response>.
Event-driven requirements	WHEN <trigger> <optional precondition>, the <system name> SHALL <system response>.
Unwanted behaviors requirements	IF <unwanted condition or undesired events>, THEN the <system name> SHALL <system response>.
State-driven requirements	WHILE <system state>, the <system name> SHALL <system response>.
Optional features requirements	WHERE <feature is included>, the <system name> SHALL <system response>.
Compound requirements	<Multiple conditions>, the <system name> SHALL <system response>.

Table 2.3: Types of EARS requirements and their patterns.

over real-world documents. As shown in Figure 2.1, an edge, is a unit of knowledge in ConceptNet that represents a particular relation between natural-language terms, or concepts. Specifically, we leverage ConceptNet to gain a better understanding of the semantic relationships between different concepts, which enables us to identify domain entities within the domain. Due to limited hardware resources, we are unable to download and build ConceptNet locally. Therefore, we use a Python library, Conceptnet-lite [4], to work with ConceptNet offline.

2.2 Related Work

2.2.1 Domain-Driven Microservice Design

In the realm of domain-driven microservice design, various studies have been conducted. Rademacher et al. [10] discussed the challenges of domain-driven design

doi:10.6342/NTU202303162

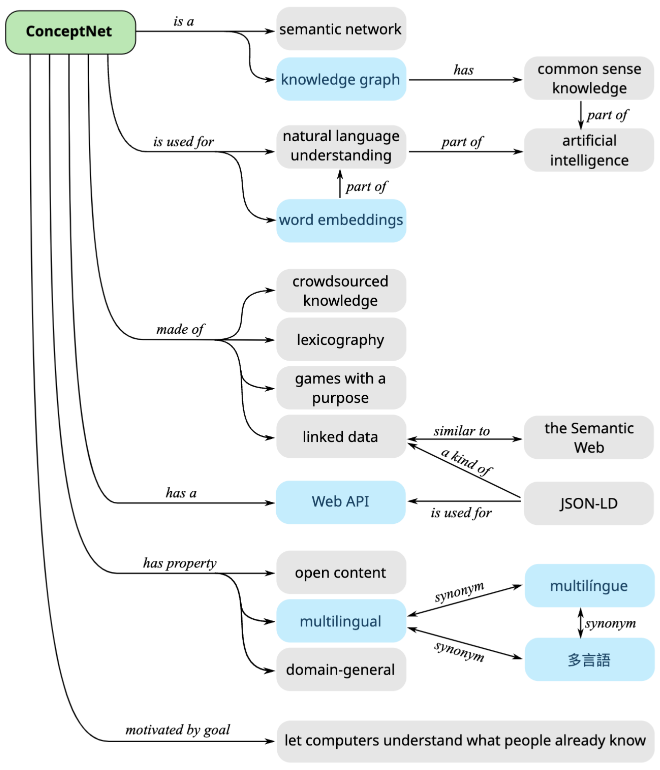


Figure 2.1: ConceptNet

doi:10.6342/NTU202303162

[6] and focused on addressing them based on model-driven development. However, during our research, we identified certain ambiguities present in domain models. As a result, we opted to design our process without relying on these models. Instead, we adopt a matchmaking mechanism to derive microservices based on the bounded contexts.

2.2.2 Microservice Granularity

Determining the appropriate size of a microservice is a complex task that involves various factors. Vural et al. [12] examined the optimal granularity for microservices based on design examples generated in previous studies that applied domain-driven design principles. Their findings highlighted that DDD has delivered favorable results for identifying modular microservices, which align well with the domain's natural boundaries.

In this work, we aim to leverage the advantages of DDD [6] to ensure that the derived microservices are cohesive. In addition, we address the granularity issue through our matchmaking mechanism. This mechanism facilitates versatile and flexible mapping between bounded context and microservices.

2.2.3 Microservices Identification

Bajaj et al. [1] proposed an approach to identifying microservices from requirements by clustering closely related use cases. While our approach shares a common foundation of utilizing use cases, we further incorporate DDD [6] concepts into our

doi:10.6342/NTU202303162

methodology By leveraging DDD principles, we aim to ensure that the resulting microservices align with the natural boundaries of the domain. Moreover, our approach employs a graph-based matchmaking mechanism that takes advantage of additional information, such as the controller, service, repository, and database schema of each microservice. This extended information allows us to better handle the complexities of microservice identification.

doi:10.6342/NTU202303162

Chapter 3

Two-Stage Process: Take An Online Bidding System as An Example

3.1 System Overview

This section provides an overview of the proposed approach that enables the deduction of microservices from requirements. As shown in Figure 3.1, the process consists of two stages. The first stage is from requirements to bounded contexts, followed by the second stage from bounded contexts to microservices. The reason behind this separation is to establish a connection between the bounded contexts and microservices without relying on the domain models, which can introduce ambiguity

doi:10.6342/NTU202303162

in the DDD [6] process.

From Requirements to Bounded Contexts The concepts of DDD are leveraged in this stage to decompose domains into contexts, which correspond to microservices that encompass distinct business logic. The process will be described in Chapter 4.

From Bounded Contexts to Microservices In this stage, a graph-based matchmaking mechanism [3] is applied to map each bounded context to its corresponding microservices. Chapter 5 will illustrate the process in detail.

Input Requirements To constrain textual requirements, we integrate the EARS requirements template [9] into our method. This integration entails requesting users to provide their input requirements specifically in the EARS format.

Output Microservices The output of the process consists of the resulting microservice candidates corresponding to each bounded context. These microservices are selected from the pool of microservice candidates [3], which serve as reusable components for building the system.

3.2 Running Example: Online Bidding System

To illustrate the process of deriving microservices from requirements, we take an online bidding system as an example. The online bidding system should provide the

doi:10.6342/NTU202303162

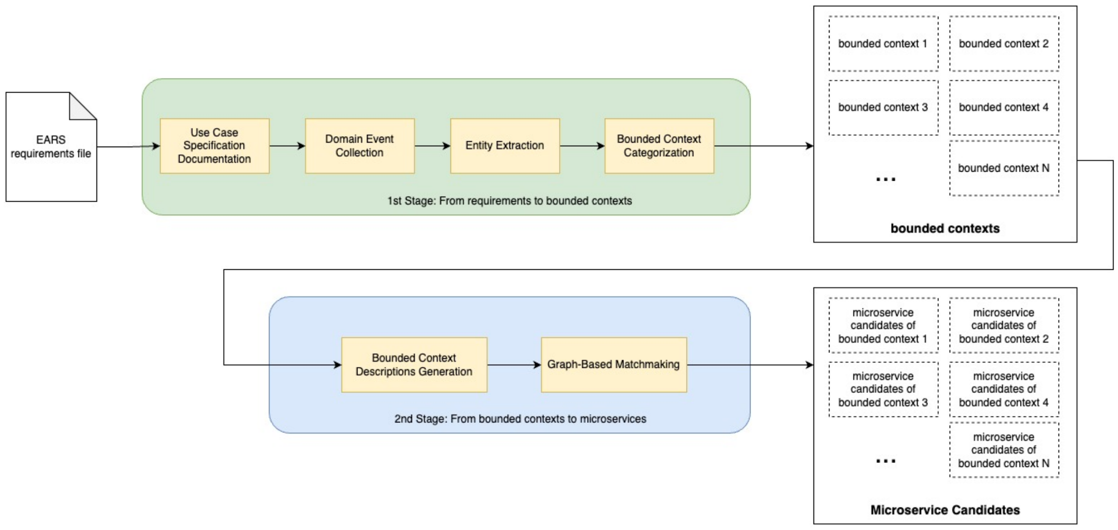

EARS requirements file
Use Case Specification Documentation
Domain Event Collection
Entity Extraction
Bounded Context Categorization
1st Stage: From requirements to bounded contexts
bounded context 1
bounded context 2
bounded context 3
bounded context 4
...
bounded context N
bounded contexts
Bounded Context Descriptions Generation
Graph-Based Matchmaking
2nd Stage: From bounded contexts to microservices
microservice candidates of bounded context 1
microservice candidates of bounded context 2
microservice candidates of bounded context 3
microservice candidates of bounded context 4
...
microservice candidates of bounded context N
Microservice Candidates

Figure 3.1: System Overview

following functionalities:

- User registration: Users, both the sellers and buyers, should register accounts to interact with the application.
- User profiles: Users can view and edit their user information.
- Item listing: Sellers can list their items for auction.
- Bidding: Buyers can place bids on the listed items.
- Notifications: The system notifies users when an auction ends, fails, or when the highest bid is updated.
- Order handling: Once an auction is won, the system manages the transaction process including payment and shipping.

doi:10.6342/NTU202303162

- Review: Buyers can provide ratings and comments for the sellers after a transaction is completed.
- Communication: Users can communicate with each other through messages.

The system's requirements are composed in the EARS format, serving as the input to the process. In this work, we further divide the requirements into five categories to differentiate their objectives:

- Front-end Functional Requirements (FFR): These describe front-end system services or functions.
- Back-end Functional Requirements (BFR): These describe back-end system services or functions.
- External Interface Requirements (EIR): These define the messages passing between subsystems and the external environment.
- Internal Interface Requirements (IIR): These define the messages passing among subsystems.
- Non-Functional Requirements (NFR): These represent constraints or goals on the system or on the development process.

Examples of the five types of requirements in the EARS format are presented in Table 3.1.

doi:10.6342/NTU202303162

Requirement Type	Example
FFR	[FFR4: Show item details] WHEN the user selects an item, the system SHALL display the selected item.
BFR	[BFR4: Update bidding information] WHILE a bid is placed, the system SHALL update the auction so that every user can see the correct information at any time.
EIR	[EIR6: User and Bidding Function Module] WHILE the user is logged in AND WHEN the bid starts, the system SHALL provide an interface for users to place a bid.
IIR	[IIR3: Database and Bid Module] WHEN the Bid Module receives the "place a bid" request, the Bid Module SHALL get the current price from the Database. IF the price of the bid is lower than the current price, THEN the Bid Module SHALL not create a new bid or update the current price.
NFR	[NFR2: Realtime updates] The system SHALL enhance the update efficiency of the bidding system and increase computing resources for price updates. IF users get outdated information, THEN the bid function module SHALL be improved.

Table 3.1: Examples of Online Bidding System EARS Requirements

doi:10.6342/NTU202303162

Chapter 4

From Requirements to Bounded Contexts

As discussed in the previous section, the first stage of the two-stage process is to derive bounded contexts from input requirements. We refer to event storming to design the process of this stage, which can be further broken down into the following four steps:

1. Write use case specifications based on EARS requirements
2. Collect domain events from use case specifications
3. Find entities
4. Bounded context categorization

First, users are required to write use case specifications following the proposed for-

doi:10.6342/NTU202303162

mat based on EARS [9] requirements. Subsequently, in line with the concepts of event storming and DDD [6], the use case specifications are used to gather domain events. The system will then extract entities from these domain events using machine learning. Lastly, the flows of each use case will be mapped to their related entities, ultimately forming the bounded contexts. The following subsections will illustrate the process involved in each of these steps.

4.1 Write Use Case Specifications Based on EARS Requirements

As Bajaj et al. [1] mentioned, use cases are widely recognized as a popular method of capturing business functional requirements. They serve as a description of the system's high-level functions and scope. However, to enable the system to parse and understand use case specifications, it is essential to ensure that they are well-formatted. Properly formatted use case specifications facilitate smoother preprocessing by providing the necessary structure and clarity for the computer to analyze and interpret the information effectively. Furthermore, some additional fields are included in use case specifications to enhance the comprehension of requirements. These additions allow us to adopt the concepts of event storming to derive bounded contexts without the need for a physical session.

The critical components of a use case specifications are participating actors, corresponding requirements, preconditions, post-conditions, basic flow of events, alternate flow, and exceptions. Additionally, we further improve the clarity of a

doi:10.6342/NTU202303162

use case specification by adding mapping between requirements and steps in basic flow. As depicted in Table 4.1, each step in the flow of a use case specification is represented by its corresponding actor, relevant EARS requirement, and the actual action performed by the actor. For example, in cases where a step is associated with an IIR, it represents the message passing among subsystems, whether it's between the frontend and the backend, between the backend subsystems, or between the backend subsystem and the database. Throughout this work, the actions related to distinct requirements are leveraged to provide information that can be exploited in different steps of the process.

4.2 Collect Domain Events From Use Case Specifications

In event storming, domain events are conventionally expressed in the past tense and adhere to the "<object> <Ved>" format. These events are organized on a timeline, enabling participants to identify the actors and commands that trigger the events.

In contrast, our approach exploits the use case specifications, which already capture the significant events of the system. These use case specifications are written in chronological order, encompassing preconditions, flows, and post-conditions. As a result, we can gather domain events directly from the descriptions provided in the use case specifications.

Moreover, in our approach, there is no need for additional steps to identify the

doi:10.6342/NTU202303162

actors and commands associated with each event, as is typically required in physical event storming sessions. The actors and commands can be readily found in the use case specifications.

In particular, for each use case specification, we collect domain events from three specific sources: the preconditions, the actions associated with BFRs, and the postconditions. We focus solely on the actions related to BFRs as they encapsulate the business logic within the domain, excluding any details related to presentation or infrastructure. We disregard the actions related to FFRs, which pertain to the behavior of the user interface and navigation. Similarly, we ignore the actions related to IIRs, as they primarily describe database operations or communication between modules. This selective approach ensures that the collected domain events align precisely with the core aspects of the domain, enhancing the relevance of the subsequent derivation process.

4.3 Find Entities

According to the definitions of DDD, domain entities represent the fundamental components, concepts, or real-world objects that play a significant role in the domain's behavior and processes. During an event storming session, participants closely analyze the events and their relationships, leading them to identify patterns and clusters of events that often indicate the presence of domain entities. Recognizing these entities can aid in defining bounded contexts within the domain.

In our research, we adopt a systematic approach that leverages machine learning

doi:10.6342/NTU202303162

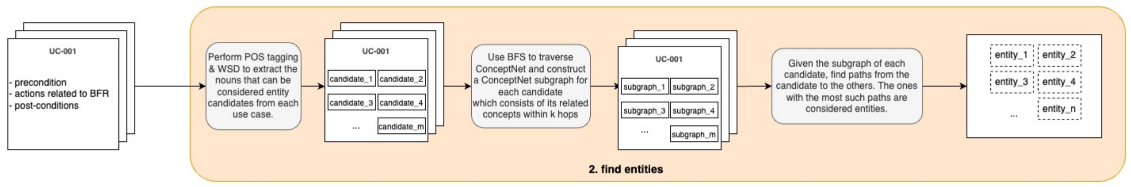

UC-001
- precondition
- actions related to BFR
- post-conditions
Perform POS tagging & WSD to extract the nouns that can be considered entity candidates from each use case.
UC-001
candidate_1
candidate_2
candidate_3
candidate_4
...
candidate_m
Use BFS to traverse ConceptNet and construct a ConceptNet subgraph for each candidate which consists of its related concepts within k hops
UC-001
subgraph_1
subgraph_2
subgraph_3
subgraph_4
...
subgraph_m
Given the subgraph of each candidate, find paths from the candidate to the others. The ones with the most such paths are considered entities.
entity_1
entity_2
entity_3
entity_4
...
entity_n
2. find entities

Figure 4.1: The process of finding entities.

to identify entities from the domain events collected from each use case specification in the previous step. Figure 4.1 depicts the process of finding entities, which is executed per use case specification. Initially, we extract nouns from the domain events that meet specific constraints, employing natural language processing (NLP) techniques such as part-of-speech (POS) tagging and word sense disambiguation (WSD). These extracted nouns are then treated as entity candidates, as they represent potential entities or their attributes.

Subsequently, we traverse ConceptNet [11], a semantic network that encompasses a wide range of commonsense concepts and relations, to comprehend each candidate's relationships with other concepts. During this step, a subgraph of ConceptNet centered around each entity candidate is constructed, providing valuable insights into their associations. Finally, we propose an algorithm to distinguish entities from all the candidates based on the information contained in the subgraphs, enabling us to derive bounded contexts in future steps.

4.3.1 Part-of-Speech (POS) Tagging

Part-of-Speech tagging is an NLP task that involves assigning a specific grammatical label or tag to each word in a given text, indicating its part of speech. We

doi:10.6342/NTU202303162

use the SpaCy [8] en_core_web_sm model to perform POS tagging on the collected domain events. Specifically, we are interested in the nouns presented within sentences, as our ultimate goal is to find entities. Entities typically refer to concepts or objects, which are commonly represented by nouns.

4.3.2 Word Sense Disambiguation (WSD)

In our work, we incorporate word sense disambiguation (WSD) as a vital step in our approach. Word sense disambiguation is a task that involves associating a word in context with the most appropriate meaning from a finite set of choices. More formally, given a word and a context, the objective is to predict the synset or sense of the word. Synsets are the groupings of synonymous words that represent the same concept.

We leverage WSD to further eliminate nouns that correspond to non-noun synsets. This process ensures that only contextually appropriate nouns are considered for further analysis. As for implementation, we use the checkpoint of the WSD model proposed by Bevilacqua et al. [2], without any fine-tuning.

4.3.3 Construct ConceptNet Subgraphs

The nouns extracted from the last step are denoted as entity candidates, as they could represent either an entity or its attributes. When users are writing descriptions of the system's desired behavior, they often mention entities and their corresponding attributes in the text. Thus, the purpose of this step is to utilize ConceptNet to

doi:10.6342/NTU202303162

gain insights into the semantic hierarchical position of a candidate. By analyzing the relationships of these candidates within ConceptNet, we can determine whether they represent entities or attributes.

For each candidate within a use case, we use breadth-first search (BFS) to traverse ConceptNet and construct a ConceptNet subgraph that includes its ”related concepts” within a specific distance, K hops in this case. In our experiments, we set this distance (K) to 2. In the subsequent steps, we disregard certain relations among the 34 defined relations in ConceptNet, as they do not contribute to distinguishing between an entity and its attributes. Table 4.2 presents a list of these ignored relations, along with the reasons for their exclusion from our approach.

In our approach, the ”related concepts” are regarded as potential attributes, and they must satisfy specific constraints. Firstly, the edges connecting them should not form a self-loop. In addition, we only consider the outward edges that start from a candidate to its related nodes, since our focus is to discover potential attributes that the entity can be associated with. However, there are two special relations, namely ”partOf” and ”derivedFrom”. The ”partOf” relation signifies a whole-part relationship between a node and its holonym, while the ”derivedFrom” relation indicates a derivation relationship. For these two relations, we consider the inward edges that start from the related nodes and connect to the candidates. In this step, the Python package NetworkX [7] is used to build the subgraphs.

doi:10.6342/NTU202303162

4.3.4 Distinguish Entities from Attributes

After building the ConceptNet subgraph of each candidate, the main goal of this step is to distinguish entities from their attributes. As illustrated in Algorithm 1, $C = \{c_1, c_2, ...c_N\}$ denotes all the candidates within a use case, and $G\{g_1, g_2, ...g_N\}$ represents their corresponding subgraphs, where N is the number of candidates within a use case. For $c_i \in C$ and $g_i \in G$, we begin by finding paths from c_i to c_j in g_i, where $j = 1, 2, ...N$ and $j \neq i$. We let $P(c_i)$ represent the number of such paths corresponding to c_i. For each use case, the candidate with the largest P is regarded as the entity, and the remaining candidates are considered its attributes. If there is no unique candidate with the maximum number of such paths, we allow users to manually select an entity from all the candidates. The user interface of this manual selection is shown in Figure 6.2. As for implementation details, we use the implementation of Dijkstra's algorithm [5] in NetworkX to find the shortest paths from a candidate to the other candidates in the subgraph.

During this process, we intentionally ignore specific words like "name", "price", "number", etc. These words are related to a vast array of concepts in ConceptNet, leading to multiple paths linking them to other candidates. Nevertheless, in most cases, these words should be viewed as attributes. Therefore, when we encounter such words, we treat them as attributes.

To demonstrate the process, we take the third use case, [UC-003 : *List an item*], as an example. In this use case, there are nine candidates: "item", "name", "description", "launch", "bidding", "photo", "reserve", "price", and "period". We have recorded the number of paths connecting each candidate to the other candidates

doi:10.6342/NTU202303162

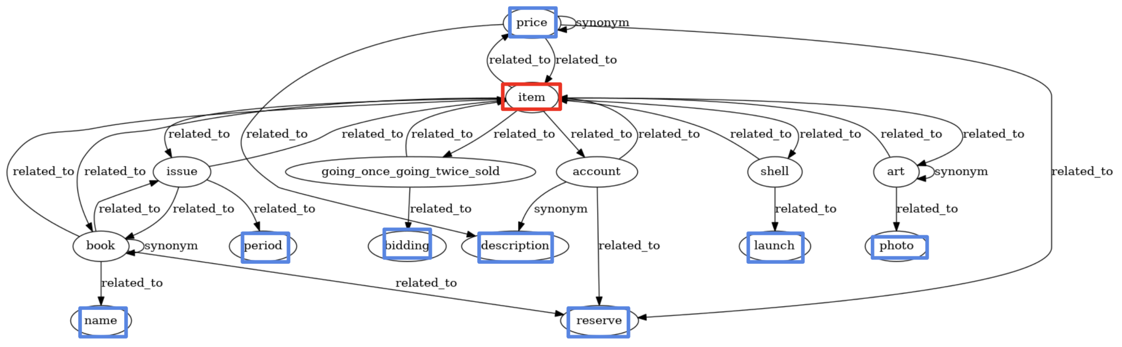


Figure 4.2: Paths to the other candidates: Item

(denoted as P) in Table 4.3. Among these nine candidates, "item" has the highest number of such paths, as depicted in Figure 4.2. As a result, it is identified as the entity for this use case.

Furthermore, there are situations where a use case lacks corresponding candidates, or when the same entity is derived from multiple use cases. In the former scenario, we exclude those use cases from the current process. However, in the latter situation, we include all the attributes of those use cases in the entity's attributes.

4.4 Bounded Context Categorization

Once the entities of the domain have been identified, we proceed to categorize the flow of use case specifications into bounded contexts. The process is described as follows:

1. Split the flow into block(s), and for each block

doi:10.6342/NTU202303162

2. extract the object(s) from the action(s) corresponding to BFR(s) and IIR(s) from backend to DB

3. compute the average ConceptNet similarity between the object(s) and each entity (from the last step)

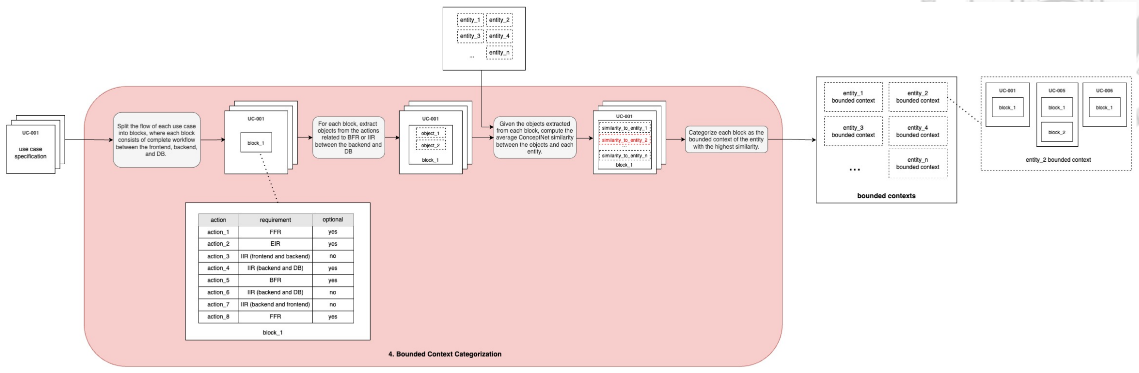
4. categorize each block into the bounded context corresponding to the entity with the highest similarity

We begin by splitting the flow of each use case specification into block(s), where a block represents a complete sub-flow that starts from an action associated with either an FFR, an EIR, or an IIR from frontend to backend and ends with an action corresponding to an IIR from backend to frontend or an FFR. For each block, we extract the objects from the actions corresponding to BFR(s) and IIR(s) from the backend to the database. This enables us to capture the relevant objects associated with the backend interactions, which indicate the underlying entities.

Next, for $o_{k,i} \in O_k = \{o_{k,1}, o_{k,2}, ...o_{k,M}\}$, where $o_{k,i}$ denotes the i-th object within the k-th block (b_k) and M denotes the number of extracted objects within b_k, we compute its similarity to the entities $E = \{e_1, e_2, ...e_T\}$ derived from the last step, where T denotes the number of entities. The term "similarity" refers to the relatedness provided by the ConceptNet API. Let $s_{k,i,j}$ denotes the similarity between $o_{k,i}$ and e_j. The average similarity of the objects within b_k to e_j, denoted as $\overline{S_{k,j}}$, can be computed as:

$$\overline{S_{k,j}} = \frac{\sum_{i=1}^{M} s_{k,i,j}}{M} \tag{4.1}$$

doi:10.6342/NTU202303162


entity_1
entity_2
entity_3
entity_4
...
entity_n
UC-001
use case specification
Split the flow of each use case into blocks, where each block consists of complete workflow between the frontend, backend, and DB.
UC-001
block_1
For each block, extract objects from the actions related to BFR or IIR between the backend and DB
UC-001
object_1
object_2
block_1
Given the objects extracted from each block, compute the average ConceptNet similarity between the objects and each entity.
UC-001
similarity_to_entity_1
similarity_to_entity_2
...
similarity_to_entity_n
block_1
Categorize each block as the bounded context of the entity with the highest similarity.
entity_1 bounded context
entity_2 bounded context
entity_3 bounded context
entity_4 bounded context
...
entity_n bounded context
bounded contexts
UC-001
block_1
UC-005
block_1
block_2
UC-006
block_1
entity_2 bounded context
block_1
4. Bounded Context Categorization

action	requirement	optional
action_1	FFR	yes
action_2	EIR	yes
action_3	IIR (frontend and backend)	no
action_4	IIR (backend and DB)	yes
action_5	BFR	yes
action_6	IIR (backend and DB)	no
action_7	IIR (backend and frontend)	no
action_8	FFR	yes

Figure 4.3: The process of categorizing bounded contexts.

We then categorize b_k into the bounded context corresponding to e_h, where

$$h = \arg\max_{j} \quad \overline{S_{k,j}} \tag{4.2}$$

This process yields a homonymous bounded context for each entity. In the next section, we will map these bounded contexts to their corresponding microservice candidates.

doi:10.6342/NTU202303162

Use Case ID	UC-003		
Use Case Name	List an item		
Goal	Achieve ”Bid on an item”		
Requirements	[FFR5: Launch an item] [BFR3: Manage an item] [EIR5: User and Item Function Module] [IIR2: Database and Item Module] [IIR6: Authentication and Item Module] [IIR13: Item Module and Item Function Module]		
Description	Sellers list an item for auction.		
Actor	Seller		
Assumptions			
Constraints			
Priority	High		
Pre-conditions	[EIR5] WHILE the user is logged in		
Post-conditions	[FFR5] The system SHALL display the listed item.		
Basic Flow	Actor	Requirement	Action
	Frontend	[FFR5]	1. The Item Function Module provides an interface for users to list an item.
	Seller	[EIR5]	2. The Seller enters the item name, description (optional), photos (optional), starting price, bidding period, and reserve price (optional) and clicks the “launch” button.
	Frontend	[FFR5]	3. The Item Function Module checks whether any required field is blank.
	Frontend	[IIR13]	4. The Item Function Module sends the $\langle\langle Create\rangle\rangle$ request with the item information the Item Module.
	Backend	[IIR6]	5. The Item Module sends the $\langle\langle Authentication\rangle\rangle$ request to the Authentication Module to check if the user is authenticated.
	Backend	[BFR3]	6. The Item Module creates an item based on the entered information.
	Backend	[IIR2]	7. The Item Module saves the item to the Database.
	Backend	[IIR13]	8. The Item Module returns the item to the Item Function Module.
	Frontend	[FFR5]	9. The Item Function Module displays the launched item.
Alternative Flow			
Exceptional Flow	3.1 One of the non-optional fields is blank.		
	Frontend	[FFR5]	3.1.1 The Item Function Module displays the message ” some fields are required” .
			3.1.2 Back to basic flow 2.
	6.1 The entered information is invalid.		
	Backend	[IIR13]	6.1.1 The Item Module returns Launched Item Fail Error back to the Item Function Module.
	Frontend	[FFR5]	6.1.2 The Item Function Module displays the “Failed to launch an item” error.
			6.1.3 Back to basic flow 2.
Use Use Case	[UC-005] View details of an Item [UC-020] Authentication		
Extend Use Case			

Table 4.1: An example of the formatted use case specification.

doi:10.6342/NTU202303162

Name	Symmetric	Reason
hasProperty	no	It connects a noun to an adjective. Outward edges for adjectives usually come with trivial relations such as synnonyms, antonyms, distinctFrom, etc., which link an adjective to another adjective.
symbolOf	no	It connects an emoji to a noun.
definedAs	no	It connects a noun to its definition, which usually does not have outward edges.
etymologicallyDerivedFrom	no	It connects two concepts from different languages. We only focus on English words.
receivesAction	no	It connects a noun to a past participle verb phrase/an adjective. A past participle usually only has inward edges with the relation ”receivesAction”.
externalURL	no	It connects a concept to a URL.
antonym	yes	We do not care about opposites.
distinctFrom	yes	We do not care about distinct members of the same set.

Table 4.2: ConceptNet relations that are ignored in this process.

candidate c_i	**item**	name	description	launch	bidding	photo	reserve	price	period
$P(c_i)$	**8**	6	6	1	1	2	4	5	5

Table 4.3: Entity candidates and their P score - [UC-003 : *List an item*]

doi:10.6342/NTU202303162

Algorithm 1 Distinguish entities from attributes

Input: $c_1 \ldots c_n \in C, g_1 \ldots g_n \in G$

Output: $entity\ (c_i)$

```
1: function FindEntity($C$, $G$)    ▷ find entity from all the candidates using their subgraphs
2:     $entity \leftarrow [\,]$
3:     $max\_num \leftarrow 0$
4:     Initialize $P(c) = 0$, for all $c \in \mathcal{C}$
5:     for $i = 1, \ldots, n$ do
6:         $num\_path \leftarrow 0$
7:         for $j = 1, \ldots, n$ do
8:             if $i == j$ then
9:                 $continue$
10:            else if findPath($g_i, c_i, c_j$) then    ▷ find paths from the current candidate to the other candidates
11:                $num\_path \leftarrow num\_path + 1$
12:            end if
13:        end for
14:        $P(c_i) \leftarrow num\_path$
15:        $max\_num \leftarrow \max(max\_num, num\_path)$
16:    end for
17:    for $i = 1, \ldots, n$ do
18:        if max_num == $P(c_i)$ then    ▷ find the candidates with the most paths from itself to the other candidates
19:            $entity.append(c_i)$
20:        end if
21:    end for
22:    if len(entity) == 1 then
23:        return $entity[0]$
24:    end if
25:    return $C$    ▷ cannot find a unique one with the most such paths, return all the candidates
26: end function

27: function findPath($G, u, v$)
28:    if there exists a path from $u$ to $v$ in $G$ then
29:        return $true$
30:    end if
31:    return $false$
32: end function
```

doi:10.6342/NTU202303162

Chapter 5

From Bounded Contexts to Microservices

In this chapter, we will discuss the process of deriving microservices from the bounded contexts generated in the previous stage. As illustrated in Figure 5.1, the process involves two main steps: the generation of bounded context descriptions and the matchmaking between these bounded context descriptions and the microservice candidates from the microservice candidate pool. The resulting mapping information will be stored in the Matchmaking DB for future reference.

5.1 Generate Bounded Context Descriptions

To perform matchmaking between bounded contexts and microservice candidates, we need to generate bounded context descriptions corresponding to the four

doi:10.6342/NTU202303162

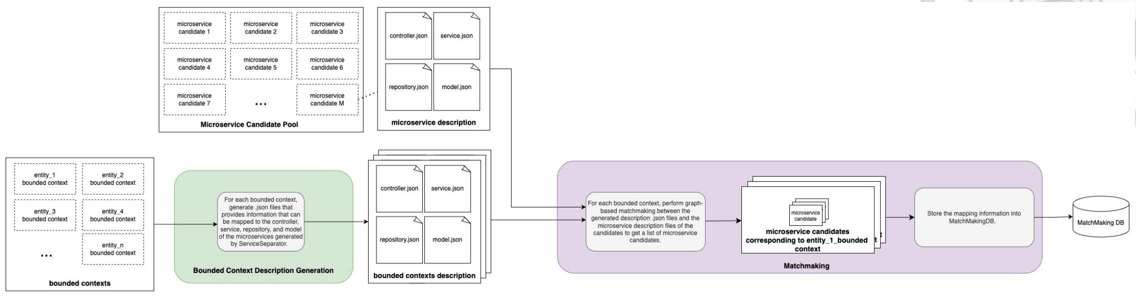

microservice candidate 1
microservice candidate 2
microservice candidate 3
microservice candidate 4
microservice candidate 5
microservice candidate 6
microservice candidate 7
...
microservice candidate M
Microservice Candidate Pool
controller.json
service.json
repository.json
model.json
microservice description
entity_1 bounded context
entity_2 bounded context
entity_3 bounded context
entity_4 bounded context
...
entity_n bounded context
bounded contexts
For each bounded context, generate .json files that provides information that can be mapped to the controller, service, repository, and model of the microservices generated by ServiceSeparator.
Bounded Context Description Generation
controller.json
service.json
repository.json
model.json
bounded contexts description
For each bounded context, perform graph-based matchmaking between the generated description .json files and the microservice description files of the candidates to get a list of microservice candidates.
microservice candidate
microservice candidates corresponding to entity_1_bounded context
Store the mapping information into MatchMakingDB.
Matchmaking
MatchMaking DB

Figure 5.1: From bounded contexts to microservices

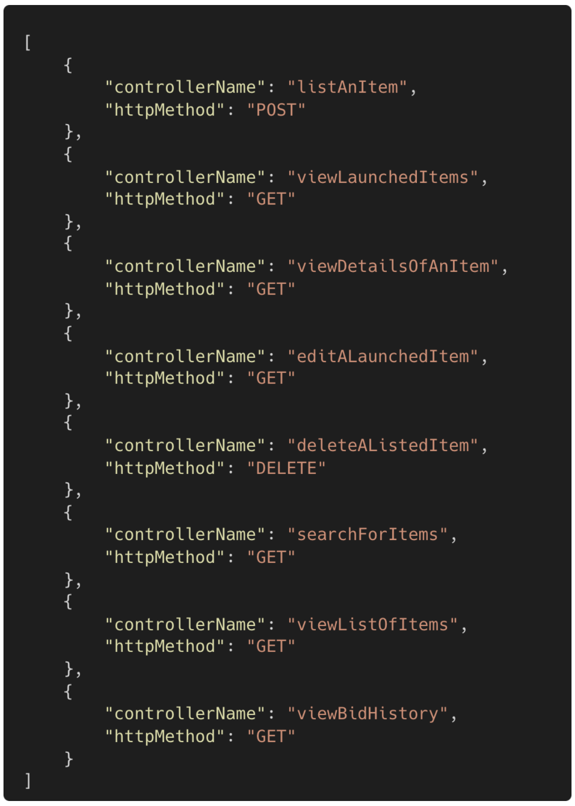
components of a microservice candidate as specified by Chen [3]:

- Controller
- Service
- Repository
- DB Schema

We take the generated descriptions of the ”item” bounded contexts as examples.

Controller Description According to Chen [3], a controller description file comprises a list of descriptions for the controller methods, which contains two fields: ”controllerName” and ”httpMethod”. However, due to limited information available in our bounded contexts, we use the use case name to fill in the ”controllerName” field, as there are usually homonymous controller methods for use cases. For the ”httpMethod” field, we leverage the stereotypes defined in our actions corresponding to IIR from the frontend to the backend: GET, POST, PUT, and DELETE, which represent the HTTP methods used for different types of requests. An example of a

doi:10.6342/NTU202303162



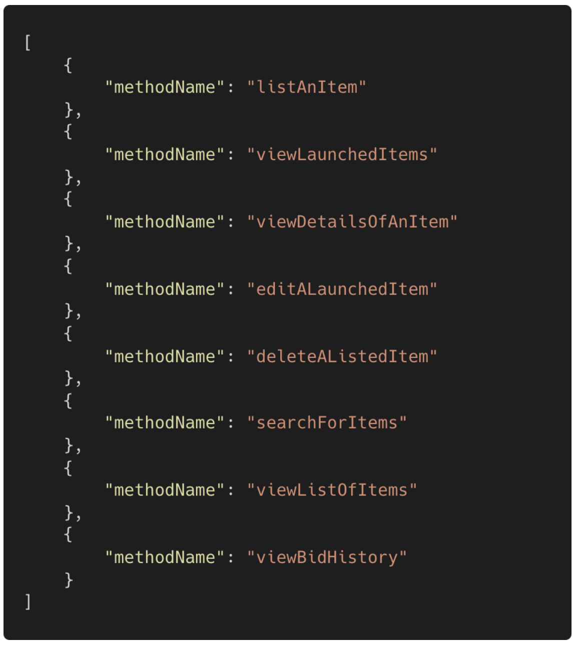
```json
[
    {
        "controllerName": "listAnItem",
        "httpMethod": "POST"
    },
    {
        "controllerName": "viewLaunchedItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewDetailsOfAnItem",
        "httpMethod": "GET"
    },
    {
        "controllerName": "editALaunchedItem",
        "httpMethod": "GET"
    },
    {
        "controllerName": "deleteAListedItem",
        "httpMethod": "DELETE"
    },
    {
        "controllerName": "searchForItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewListOfItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewBidHistory",
        "httpMethod": "GET"
    }
]
```

Figure 5.2: Item bounded context - controller description JSON file

controller description is presented in Figure 5.2, where the use case name is used to identify the controller, and the appropriate HTTP method is assigned based on the stereotype associated with the corresponding IIR.

Service Description A service description file comprises a list of descriptions for the service methods, which include a field, called ”methodName” which represents the name of a service method. Since in the Model-View-Controller architecture, the Service class acts as an intermediary between the Model and the Controller, we observe that in the naming of the service methods, it is common practice to use the

doi:10.6342/NTU202303162



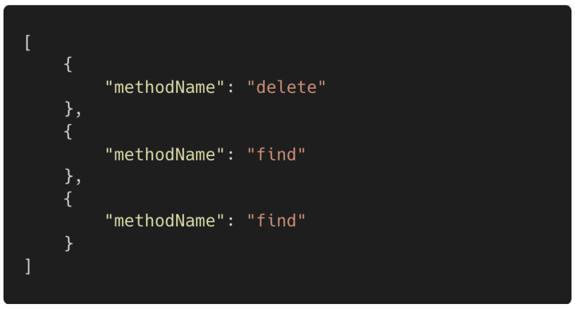
```
[
    {
        "methodName": "listAnItem"
    },
    {
        "methodName": "viewLaunchedItems"
    },
    {
        "methodName": "viewDetailsOfAnItem"
    },
    {
        "methodName": "editALaunchedItem"
    },
    {
        "methodName": "deleteAListedItem"
    },
    {
        "methodName": "searchForItems"
    },
    {
        "methodName": "viewListOfItems"
    },
    {
        "methodName": "viewBidHistory"
    }
]
```

Figure 5.3: Item bounded context - service description JSON file

same name for the service methods as the corresponding controller methods. Given this convention, when generating the service description file, we also use the use case name to fill in the ”methodName” field. An example of a service description is presented in Figure 5.3.

Repository Description A repository description file comprises a list of descriptions for the repository methods, which include a field, called ”methodName”, which represents the name of a repository method. To generate descriptions based on our derived bounded contexts, we leverage the stereotypes defined in the actions associated with IIRs between the backend and the database. Examples of such stereotypes may include ”find,” ”delete,” ”save,” and others. Each stereotype corresponds to a

doi:10.6342/NTU202303162



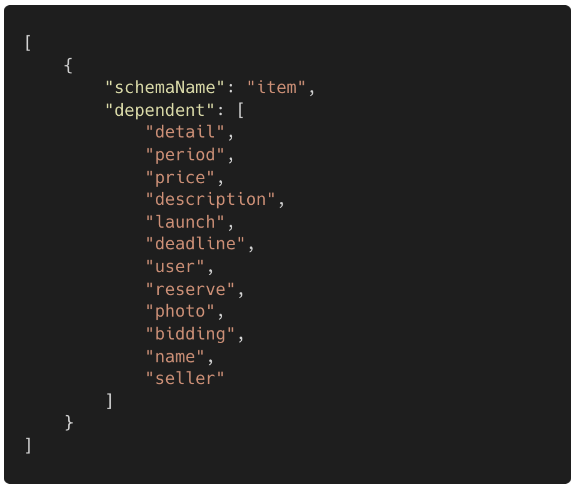
```
[
    {
        "methodName": "delete"
    },
    {
        "methodName": "find"
    },
    {
        "methodName": "find"
    }
]
```

Figure 5.4: Item bounded context - repository description JSON file

specific type of operation that the system can perform when interacting with the database. An example of a repository description is presented in Figure 5.4.

DB Schema Description A database schema description file comprises a list of descriptions for the data models, each of which includes two fields: ”schemaName” and ”dependent”. In our approach, we leverage the information gathered during the entity extraction process to map to the fields in the database schema description file. The entities we derive from the domain events serve as the data models, and their names are used to populate the ”schemaName” field in the database schema description. Furthermore, the attributes we identify for each entity correspond to the related tables in the database. As a result, we populate the ”dependent” field in the database schema description with the names of the attributes that are associated with each data model. An example of a repository description is presented in Figure 5.5.

doi:10.6342/NTU202303162



```
[
    {
        "schemaName": "item",
        "dependent": [
            "detail",
            "period",
            "price",
            "description",
            "launch",
            "deadline",
            "user",
            "reserve",
            "photo",
            "bidding",
            "name",
            "seller"
        ]
    }
]
```

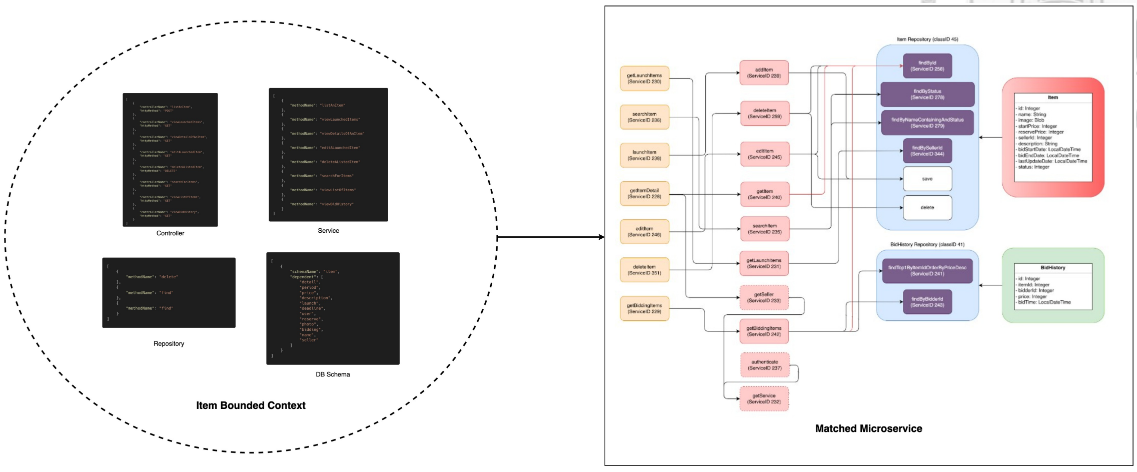
Figure 5.5: Item bounded context - DB schema description JSON file

5.2 Get Microservice Candidates by Matchmaking

Once the bounded context descriptions have been generated, we proceed to submit a request to the matchmaking server [3] for each bounded context. These requests include the corresponding bounded context description files. The matchmaking server then performs a graph-based matchmaking algorithm and returns a list of microservice candidates ranked by their similarities.

In particular, when the matchmaking server receives a request containing the bounded context descriptions, it proceeds to construct a graph for each of the four descriptions provided in the request. Simultaneously, the server also retrieves the microservice candidates from the microservice candidate pool. For each microservice candidate, it constructs four graphs, corresponding to the four components of the

doi:10.6342/NTU202303162



Controller

```
[
    {
        "controllerName": "listAnItem",
        "httpMethod": "POST"
    },
    {
        "controllerName": "viewLaunchedItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewDetailsOfAnItem",
        "httpMethod": "GET"
    },
    {
        "controllerName": "editALaunchedItem",
        "httpMethod": "GET"
    },
    {
        "controllerName": "deleteAListedItem",
        "httpMethod": "DELETE"
    },
    {
        "controllerName": "searchForItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewListOfItems",
        "httpMethod": "GET"
    },
    {
        "controllerName": "viewBidHistory",
        "httpMethod": "GET"
    }
]
```

Service

```
[
    {
        "methodName": "listAnItem"
    },
    {
        "methodName": "viewLaunchedItems"
    },
    {
        "methodName": "viewDetailsOfAnItem"
    },
    {
        "methodName": "editALaunchedItem"
    },
    {
        "methodName": "deleteAListedItem"
    },
    {
        "methodName": "searchForItems"
    },
    {
        "methodName": "viewListOfItems"
    },
    {
        "methodName": "viewBidHistory"
    }
]
```

Repository

```
[
    {
        "methodName": "delete"
    },
    {
        "methodName": "find"
    },
    {
        "methodName": "find"
    }
]
```

DB Schema

```
[
    {
        "schemaName": "item",
        "dependent": [
            "detail",
            "period",
            "price",
            "description",
            "launch",
            "deadline",
            "user",
            "reserve",
            "photo",
            "bidding",
            "name",
            "seller"
        ]
    }
]
```

Item Bounded Context
getLaunchItems (ServiceID 230)
searchItem (ServiceID 236)
launchItem (ServiceID 238)
getItemDetail (ServiceID 228)
editItem (ServiceID 246)
deleteItem (ServiceID 351)
getBiddingItems (ServiceID 229)
addItem (ServiceID 239)
deleteItem (ServiceID 259)
editItem (ServiceID 245)
getItem (ServiceID 240)
searchItem (ServiceID 235)
getLaunchItems (ServiceID 231)
getSeller (ServiceID 233)
getBiddingItems (ServiceID 242)
authenticate (ServiceID 237)
getService (ServiceID 232)
Item Repository (classID 45)
findById (ServiceID 258)
findByStatus (ServiceID 278)
findByNameContainingAndStatus (ServiceID 279)
findBySellerId (ServiceID 344)
save
delete
Item
- id: Integer
- name: String
- image: Blob
- startPrice: Integer
- reservePrice: Integer
- sellerId: Integer
- description: String
- bidStartDate: LocalDateTime
- bidEndDate: LocalDateTime
- lastUpdateDate: LocalDateTime
- status: Integer
BidHistory Repository (classID 41)
findTop1ByItemIdOrderByPriceDesc (ServiceID 241)
findByBidderId (ServiceID 243)
BidHistory
- id: Integer
- itemId: Integer
- bidderId: Integer
- price: Integer
- bidTime: LocalDateTime
Matched Microservice

Figure 5.6: The microservice with the highest similarity to the Item Bounded Context.

microservice. Next, the server performs the alignment process, where it matches the nodes of the request graph with the nodes of each microservice candidate graph for each component. After the alignment, the server proceeds to compute the similarity between the graph corresponding to the request and each microservice candidate graph. As a result, it yields four similarities for each microservice candidate, corresponding to the four components. To aggregate the similarities from the four components, the server calculates a weighted similarity score for each microservice candidate. Finally, it returns a sorted list of microservice candidates.

To demonstrate, figure Figure 5.6 illustrates the microservice candidate with the highest similarity to the Item Bounded Context.

doi:10.6342/NTU202303162

Chapter 6

User Interface

To enable seamless interaction with our system, we develop a front-end client that controls the process flow. With this front-end interface, users can upload files as input to the process, edit the intermediate entity results, and view the matched microservice candidates.

Upload Requirements and Use Case Specifications To adopt our proposed approach, users are required to upload system requirements in EARS [9] format and the formatted use case specification. As shown in Figure 6.1, we have developed an interface for uploading files.

Edit Entity List As described in Section 4.3, there exist scenarios where the system faces challenges in distinguishing entities from all the candidates. To address this, we have developed an intuitive interface that allows users to manually select

doi:10.6342/NTU202303162

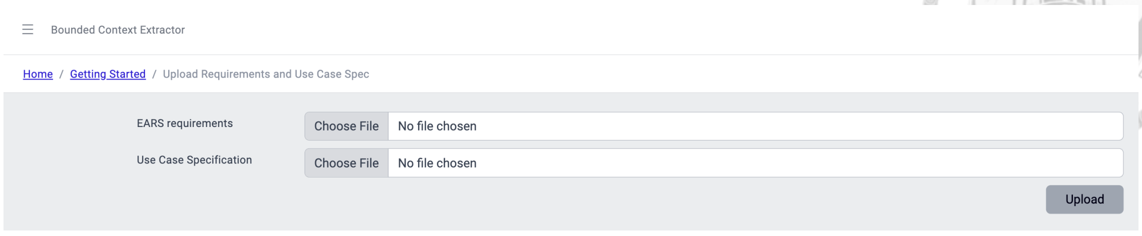
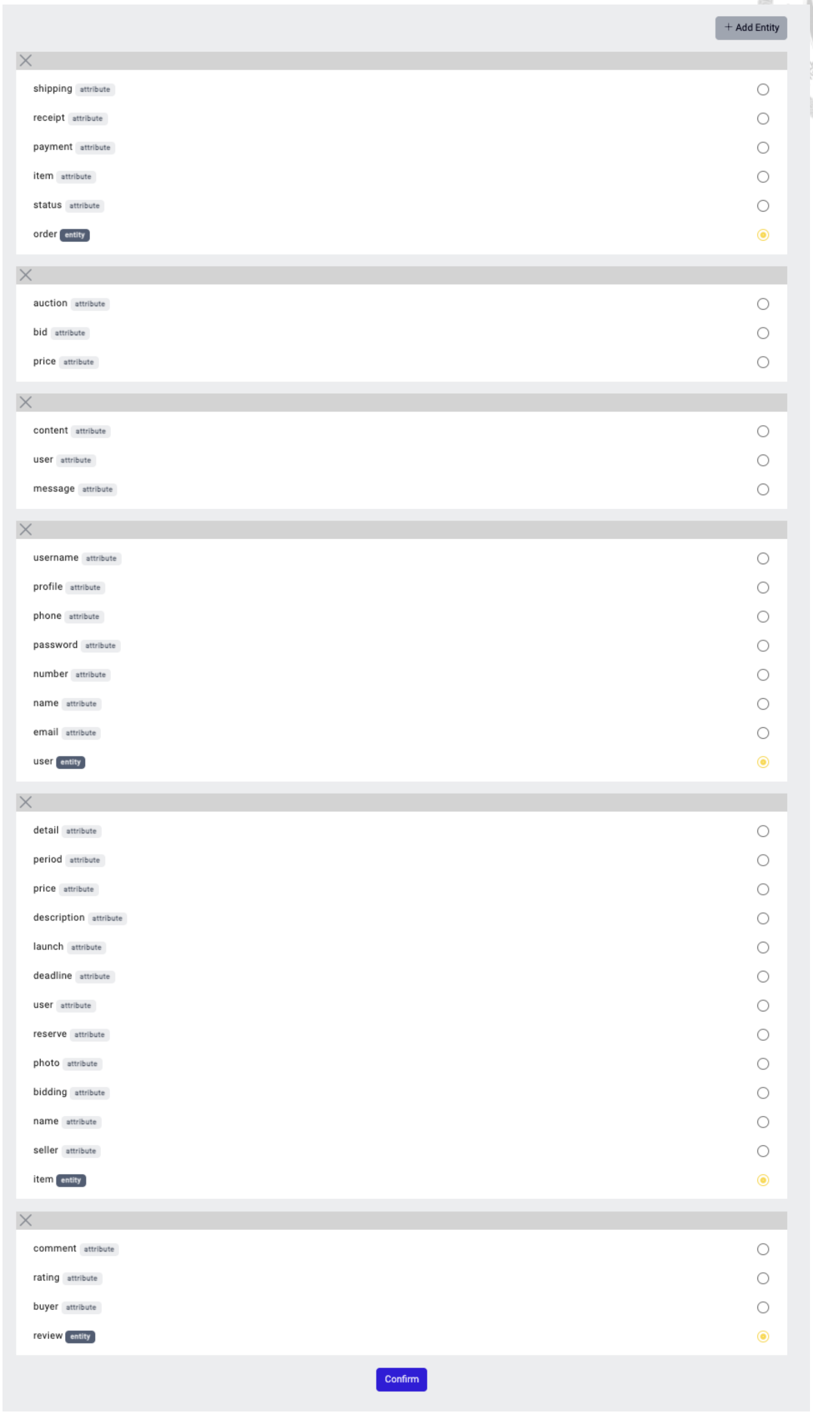


Figure 6.1: The UI for uploading requirements and use case specifications

entities when needed. Moreover, when the system does not extract all the desired entities successfully, users have the option to use the interface to specify their expected entities. This ensures that essential entities are not overlooked, facilitating a more accurate process. The interface is shown in Figure 6.2.

Display Resulting Microservice Candidates As discussed in Section 5.2, the matchmaking server [3] will return a list of microservice candidates corresponding to each bounded context based on the received requests. These microservice candidates are ranked according to their similarity to the corresponding bounded context. To provide users with valuable insights, we have developed an interface that showcases the matchmaking results. Upon accessing the interface, users can explore the list of microservice candidates associated with each bounded context. The interface allows users to delve deeper into individual microservice candidates to explore their characteristics.

doi:10.6342/NTU202303162



×

- shipping attribute ○
- receipt attribute ○
- payment attribute ○
- item attribute ○
- status attribute ○
- order entity ◉

×

- auction attribute ○
- bid attribute ○
- price attribute ○

×

- content attribute ○
- user attribute ○
- message attribute ○

×

- username attribute ○
- profile attribute ○
- phone attribute ○
- password attribute ○
- number attribute ○
- name attribute ○
- email attribute ○
- user entity ◉

×

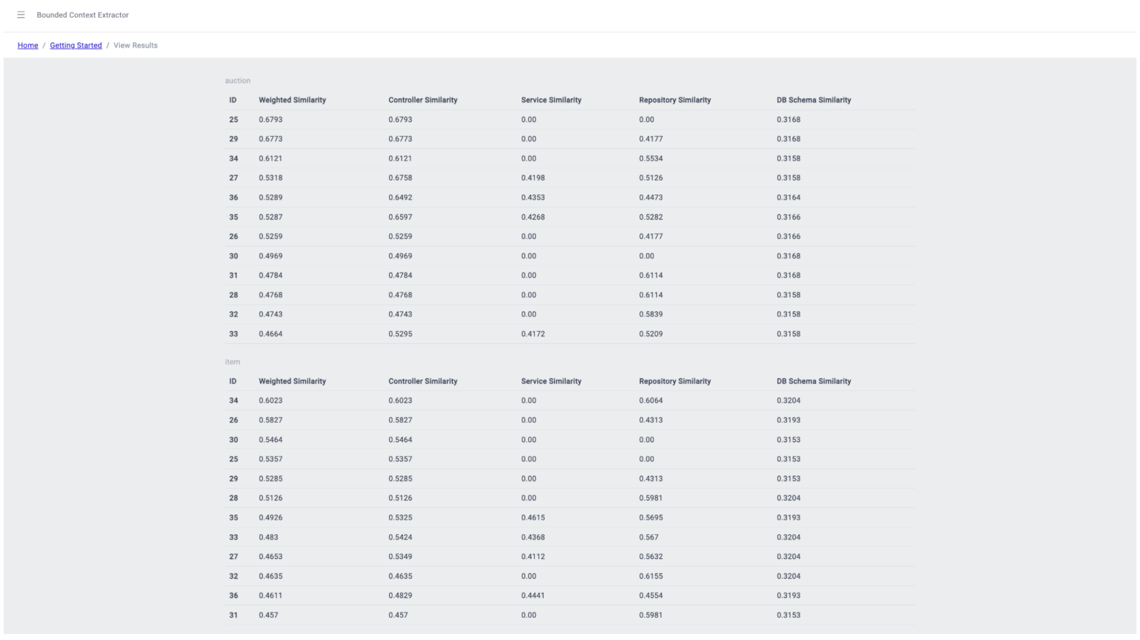
- detail attribute ○
- period attribute ○
- price attribute ○
- description attribute ○
- launch attribute ○
- deadline attribute ○
- user attribute ○
- reserve attribute ○
- photo attribute ○
- bidding attribute ○
- name attribute ○
- seller attribute ○
- item entity ◉

×

- comment attribute ○
- rating attribute ○
- buyer attribute ○
- review entity ◉

Figure 6.2: The UI for editing entity list

doi:10.6342/NTU202303162


☰ Bounded Context Extractor
Home / Getting Started / View Results

auction

ID	Weighted Similarity	Controller Similarity	Service Similarity	Repository Similarity	DB Schema Similarity
25	0.6793	0.6793	0.00	0.00	0.3168
29	0.6773	0.6773	0.00	0.4177	0.3168
34	0.6121	0.6121	0.00	0.5534	0.3158
27	0.5318	0.6758	0.4198	0.5126	0.3158
36	0.5289	0.6492	0.4353	0.4473	0.3164
35	0.5287	0.6597	0.4268	0.5282	0.3166
26	0.5259	0.5259	0.00	0.4177	0.3166
30	0.4969	0.4969	0.00	0.00	0.3168
31	0.4784	0.4784	0.00	0.6114	0.3168
28	0.4768	0.4768	0.00	0.6114	0.3158
32	0.4743	0.4743	0.00	0.5839	0.3158
33	0.4664	0.5295	0.4172	0.5209	0.3158

item

ID	Weighted Similarity	Controller Similarity	Service Similarity	Repository Similarity	DB Schema Similarity
34	0.6023	0.6023	0.00	0.6064	0.3204
26	0.5827	0.5827	0.00	0.4313	0.3193
30	0.5464	0.5464	0.00	0.00	0.3153
25	0.5357	0.5357	0.00	0.00	0.3153
29	0.5285	0.5285	0.00	0.4313	0.3153
28	0.5126	0.5126	0.00	0.5981	0.3204
35	0.4926	0.5325	0.4615	0.5695	0.3193
33	0.483	0.5424	0.4368	0.567	0.3204
27	0.4653	0.5349	0.4112	0.5632	0.3204
32	0.4635	0.4635	0.00	0.6155	0.3204
36	0.4611	0.4829	0.4441	0.4554	0.3193
31	0.457	0.457	0.00	0.5981	0.3153

Figure 6.3: The UI for displaying resulting microservice candidates

doi:10.6342/NTU202303162

Chapter 7

Conclusion

In this paper, we propose a two-stage process that derives microservices from requirements. The process aids in defining bounded contexts in a systematic approach. Moreover, the process guarantees that the connections between bounded contexts and microservices are adaptable and versatile. By applying the concepts of DDD [6], we ensure that the mapped microservices align with the business processes of a system.

doi:10.6342/NTU202303162

Chapter 8

Future Work

In the future, we plan to automate the entire process without any human intervention. Currently, there are certain situations where the system faces challenges in distinguishing entities from noun candidates. For this reason, users are required to manually select the entities. Additionally, the generated bounded context descriptions provide limited information that can be used for matchmaking microservices. We are convinced that there is still untapped potential in the existing data, and there might be valuable information that has not been included in the current process. Therefore, our focus will be on further exploring the requirements to enhance the bounded context descriptions, making them more informative.

As part of our future work, we intend to explore the potential of domain models. Despite the ambiguity in deducing microservices from them, we believe that they remain valuable assets in deriving microservices, as they represent the abstraction of the business domain. Consequently, we will also investigate the systematic design

doi:10.6342/NTU202303162

of domain models and their mapping to microservices.

doi:10.6342/NTU202303162

Bibliography

[1] D. Bajaj, A. Goel, and S. C. Gupta. Greenmicro: Identifying microservices from use cases in greenfield development. *IEEE Access*, 10:67008–67018, 2022.

[2] M. Bevilacqua and R. Navigli. Breaking through the 80% glass ceiling: Raising the state of the art in word sense disambiguation by incorporating knowledge graph information. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2854–2864, Online, July 2020. Association for Computational Linguistics.

[3] L.-S. Chen. From monolithic to microservice: A dependency decoupling approach. Master's thesis, National Taiwan University, 2023.

[4] Conceptnet-lite. `https://github.com/ldtoolkit/conceptnet-lite`.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[6] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[7] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, and J. Millman, editors,

doi:10.6342/NTU202303162

Proceedings of the 7th Python in Science Conference, pages 11 – 15, Pasadena, CA USA, 2008.

[8] M. Honnibal and I. Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[9] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy approach to requirements syntax (ears). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322, 2009.

[10] F. Rademacher, J. Sorgalla, and S. Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3):36–43, 2018.

[11] R. Speer, J. Chin, and C. Havasi. Conceptnet 5.5: An open multilingual graph of general knowledge, 2018.

[12] H. Vural and M. Koyuncu. Does domain-driven design lead to finding the optimal modularity of a microservice? *IEEE Access*, 9:32721–32733, 2021.

doi:10.6342/NTU202303162